VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Detekce objektů v mračnu 3D bodů

Recognizing Objects in 3D Point Clouds

2018 Martin Vaněk

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Vaněk**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Detekce objektů v mračnu 3D bodů
Recognizing Objects in 3D Point Clouds

Jazyk vypracování: čeština

Zásady pro vypracování:

Mračna bodů představují jednoduchou metodu pro zobrazení a interpretaci měření objektů v terénu. Body jsou získány například pomocí rekonstrukce scény z dvojice stereokamer nebo laserovým snímačem (např. Velodyne LiDAR). Cílem diplomové práce je navrhnout postup pro automatickou registraci a mapování 3D prostoru z mračna získaných bodů. Výsledná aplikace by měla být schopna vytvořit prostorový model scény a detekovat objekty v zájmové oblasti.

Řešení bude obsahovat:
1. Popis stávajících přístupů pro zpracování mračen 3D bodů, SLAM algoritmy.
2. Podrobný popis navrženého řešení.
3. Implementace algoritmu pro mapování 3D prostoru s jednoduchou detekci objektů.
4. Integraci navrženého algoritmu se senzorem Velodyne LiDAR.
4. Testování implementace na volně dostupných datových kolekcí a datech nasbíraných v areálu VŠB.
5. Závěr a vyhodnocení dosažených výsledků.

Seznam doporučené odborné literatury:

[1] Song Zhang, Handbook of 3D Machine Vision: Optical Metrology and Imaging. CRC Press. 2013. ISBN 1439872198
[2] Richard Szeliski, Computer Vision: Algorithms and Applications, Springer, 2011. ISBN 1848829345
[3] Richard Hartley, Multiple View Geometry in Computer Vision. Cambridge University Press. 2004. ISBN 0521540518

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Ing. Michal Krumnikl, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2018 .....

Tímto bych chtěl poděkovat Mgr. Ing. Michalu Krumniklovi, Ph.D. za jeho trpělivost a ochotu se mnou řešit veškeré problémy, které nastaly během práce. Bez něj by tato rozhodně práce nevznikla. Dále bych chtěl poděkovat svojí rodině za podporu a obzvláště svému bratrovi za jeho cenné rady.

Abstrakt

Samořídicí roboti a auta se v posledních letech stali horkým tématem široké veřejnosti. Aby fungovalo plánování cesty a detekce překážek, tak robot musí znát mapu prostředí a jeho polohu v této mapě. V neznámém prostředí takovou mapu nemá a tím pádem v ní polohu určit nemůže. Proto robot musí mapu tvořit za jízdy a zároveň se v ní musí umět najít. Takovou situaci řeší současné mapování a lokalizace, zkráceně SLAM. Uvnitř práce se dozvíte, co SLAM znamená a jak se dá řešit. Pro samotné tvoření mapy se v této práci využívá laserový snímač Velodyne LiDAR, jehož výstupem je mračno bodů v prostoru. Tento výstup zpracovává open-source software *Google Cartographer*. V práci je popsána jeho konfigurace a následně pro výslednou mapu je implementován detektor překážek.

Klíčová slova: LiDAR, lokalizace, mapování, SLAM, detekce překážek, ROS

Abstract

Self-driving cars and robots have became a hot topic for a broad audience in a recent years. Robot has to know map of the environment and their location in such a map for proper path planning and obstacle detection. In unknown environment the robot doesn't have the map so it can't find itself in it. This situation is solved by Simultaneous Localization and Mapping, which is a long name for SLAM. In this thesis you will learn what SLAM actually means and how to solve it. Laser sensor Velodyne LiDAR produces point cloud and is used for creating the map. Point cloud is fed into open-source software *Google Cartographer*. In the thesis its configuration is described and obstacle detector is implemented for the created map.

Key Words: LiDAR, localization, mapping, SLAM, obstacle detection, ROS

Obsah

Seznam použitých zkratek a symbolů **9**

Seznam obrázků **10**

Seznam výpisů zdrojového kódu **11**

1 Úvod **12**

2 Popis přístupů SLAM **13**
2.1 Lidarová technologie 13
2.2 Důležité pojmy 14
2.3 SLAM - Simultaneous Localisation and Mapping 15

3 Použitá zařízení a technologie **26**
3.1 Velodyne VLP-16 26
3.2 Robot Operating System 27
3.3 Google Cartographer 31
3.4 Octomap 33
3.5 Point Cloud Library 33

4 Konfigurace a instalace **34**
4.1 Seznámení s VLP-16 34
4.2 Problém s 3D reprezentací 36
4.3 Seznámení a instalace Cartographeru 37

5 Implementace **43**
5.1 Získávání 3D dat z Cartographeru 43
5.2 Implementace balíčku obstacle_detection 44
5.3 Vizualizace 48

6 Testování **49**
6.1 Popis testovacího prostředí 49
6.2 Testovací sady 50

7 Závěr **52**

8 Literatura **53**

Přílohy **55**

A Obsah přiloženého DVD **56**

B První konfigurace pro spuštění Cartographeru **57**

C Testovací prostředí **60**

D Výsledky testovacích případů **61**

Seznam použitých zkratek a symbolů

EKF	–	Extended Kalman Filter
GPS	–	Global Positioning System
ICP	–	Iterative Closest Point
IMU	–	Inertial measurement unit
LiDAR	–	Light Detection And Ranging
PCL	–	Point Cloud Library
Póza	–	Poloha a orientace
ROS	–	Robot Operating System
SLAM	–	Simultaneous localization and mapping
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
URDF	–	Unified Robot Description Format
URI	–	Uniform Resource Identifier

Seznam obrázků

1 Převod polohy 3D bodu z úhlů paprsků a vzdálenosti do formátu XYZ převzato z dokumentace pro VLP-16 [5] . . . 14
2 Ukázka fungování SLAM algoritmu založeném na Kalmanově filtru, převzato z [17] 18
3 Ukázka principu Importance Sampling . . . 21
4 Ukázka grafu pro algoritmus GraphSLAM . . . 24
5 Velodyne Puck™ VLP-16 (převzato z [5]) . . . 26
6 ROS diagram posílání zpráv . . . 28
7 ROS diagram volání služby . . . 28
8 Vizualizace 3D mračna pomocí *Rvizu* . . . 30
9 Vizualizace `bag` souboru pomocí *rqt_bag* . . . 30
10 Přehled systému Cartographer (převzato z [27]) . . . 32
11 Ukázka VeloView s daty ze školního senzoru VLP-16 na systému Windows . . . 34
12 Sekvenční diagram, jenž naznačuje integraci dat ve vlastním balíčku, odesílané *Cartographerem* skrze *ROS* . . . 46
13 Strom transformací při spuštěném *Cartographeru* a mnou implementovaným uzlem 47
14 Vizualizace kolize a 3D mapy v programu *Rviz* . . . 48
15 Blokové schéma testovací prostředí . . . 49
16 Výsledek mapování druhého patra na budově FEI . . . 51
17 Fotka našeho auta, ze kterého byly pořizovány testovací data. Autor fotky: Mgr. Ing. Michal Krumnikl, Ph.D. . . . 60
18 Výsledek mapy při pohledu shora nad záznamem `dod.bag` . . . 61
19 Výsledek mapy při pohledu shora nad záznamem `druhe_patro_1.bag` . . . 61
20 Výsledek mapy při pohledu shora nad záznamem `druhe_patro_2.bag` . . . 62

Seznam výpisů zdrojového kódu

1 Výpis Cartographeru při problému s telefonem IMU 39
2 Úprava zdrojového kódu třídy ImuPublisher v aplikaci ROS Sensors drivers . . . 39
3 Příkazy potřebné pro autorizaci ROS pro vzdálené spuštění 41
4 Definice služby SubmapCloudQuery 43
5 Konfigurace `.launch` souboru pro správné spuštění *Cartographeru* nad `bag` záznamem 57
6 Popis robota ve formátu urdf pro spuštění *Cartographeru* 58
7 Konfigurace *Cartographeru* 59

1 Úvod

Samořiditelná auta a roboti se dostávají v posledních letech do povědomí široké veřejnosti čím dál víc. Jedním z důvodů je zvýšení dostupnosti takových vozidel a postupné úpravy zákonů pro jejich povolení. V roce psaní této práce samořiditelné auto částečně zavinilo první smrtelnou nehodu[1] a i proto je třeba se zabývat detekcí překážek při jízdě.

Pojďme se podívat, na jednu část samořiditelného vozidla. Aby auto bylo plně autonomní, potřebuje mít nějakou reprezentaci okolního prostředí, nějakou mapu a to i v místech, kde není satelitní signál. Dále, aby fungovaly plánovače cest a detektory překážek, musí znát aktuální polohu a natočení v mapě. Této dvojici se říká póza a bude v práci často zmiňovaná. Pokud samořiditelné auto vjede do oblasti, kterou nezná, musí být schopné si tvořit mapu samo a určit v ní svoji polohu, co kdybychom později potřebovali jet zpět?

Současnému tvoření mapy a zjišťování polohy se říká *SLAM* (zkratka pro *Simultaneous Location And Mapping*) a jeho první zmínky se datují od roku 1990, kdy byly vydány první výzkumné publikace definující tento problém [1] a [2]. Během vývoje vznikly tři rodiny řešení tohoto problému. Všechny si v této práci popíšeme a naznačíme jak fungují jejich populární implementace. Nepůjde o kompletní výpis, jako spíše o pochopení jejich hlavní myšlenky.

V práci jsem použil *Google Cartographer* [3], který implementuje nejnovější přístup řešení problému *SLAM*. *Google Cartographer* existuje jako balíček do frameworku *Robot Operating system* (dále jen *ROS*), což je open-source framework, který poskytuje prostředí pro propojování již hotových komponent k vytvoření softwaru robota, ale dá se použít i pro ne-robotické věci. Obsahuje hotové nástroje pro vizualizaci, debugging, logování, přenos dat po síti, záznam do uniformního formátu a mnoho dalšího. V práci si popíšeme jak *ROS*, tak *Cartographer*.

Google Cartographer není knihovna, která funguje „out of the box“, vyžaduje specifické nastavení a zařízení, které zde budou popsány. V době psaní této práce neexistoval kompatibilní framework pro detekci kolizí v 3D či autonomní plánování s výstupem od *Google Cartographeru*. Cílem této práce bylo napsat knihovnu pro detekci překážek v jeho vytvořené 3D mapě. Na závěr se podíváme, jak vypadá výsledek mapy vytvořené jízdou na nové budově Fakultě Elektrotechniky a Informatiky.

[1] https://www.novinky.cz/auto/467003-selhalo-uplne-vsechno-technika-i-ridic-hodnoti-odbornik-smrtelnou-nehodu-samoriditelneho-auta.html

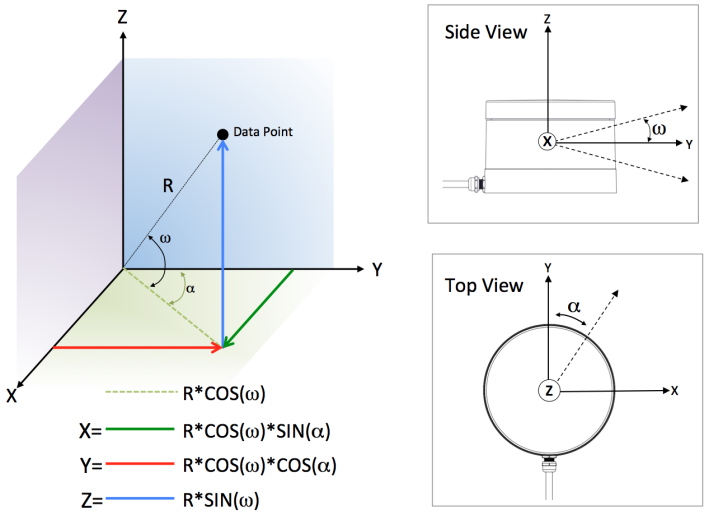
2 Popis přístupů SLAM

Čtenář se v této kapitole prvně seznámí s technologií *LiDAR* (zkratka pro *Light Detection And Ranging*, dále pouze lidar), která se používá pro velmi přesné měření prostoru a využívá se i v praktické části práce. Výstupem měření 3D lidaru, je velké množství bodů v prostoru, čemuž se říká mračno bodů. Mračno bodů mapuje prostor v jednom místě, nicméně pokud máme robota (v této práci byl použit model auta, ale může to být cokoliv, klidně i člověk, který nese lidarový senzor), který se pohybuje a jeho cílem je vytvořit mapu celého prostředí, musí se mračna nějakým způsobem spojit. Robot musí znát svoji pózu, aby věděl, kam do mapy umístit získaná data. Na druhou stranu, aby ji mohl zjistit, musí znát mapu, což vytváří problém typu „chicken and egg“[2]. Takový problém řeší *SLAM* a v této sekci bude podrobněji vysvětlen. Následně zde budou popsány tři rodiny *SLAM* řešení. V této práci byl využíván software *Cartographer*, který je z rodiny řešení na základě optimalizace grafu (sekce 2.3.3), ostatní rodiny jsou zde pro porovnání a u každé z nich bude naznačena implementace jednoho algoritmu společně s vysvětlením.

2.1 Lidarová technologie

Light Detection And Ranging označuje technologii, která umí změřit vzdálenosti od povrchu pomocí paprsku světla. Často se k tomu využívají světelné paprsky v blízké infračervené oblasti elektromagnetické spektra. Vzdálenost se zjistí měřením času, který paprsek strávil na cestě k a zpět od povrchu. Lidar je obdobou radaru (radio detecting and ranging), kde místo rádiových vln se posílají pulzy světla. Souřadnice bodu v prostoru se vypočítají pomocí změřené vzdálenosti, pózy senzoru (pokud ji neznáme, můžeme brát, že je senzor středem souřadnicové soustavy) a úhlu pod kterým byl paprsek vyslán [4]. Ukázku převodu můžeme vidět na obrázku č. 1, kde ω a α jsou úhly pod kterým je paprsek vyslán a R je změřená vzdálenost.

[2] označuje otázku co bylo dřív kuře nebo vejce



Obrázek 1: Převod polohy 3D bodu z úhlů paprsků a vzdálenosti do formátu XYZ převzato z dokumentace pro VLP-16 [5]

Lidar vysílá paprsky vysokou frekvencí (např. 150 tisíc pulzů za sekundu) a výsledné 3D body tvoří mračno bodů. Lidary můžou být i jednodušší senzory, které měří vzdálenosti na jedné ploše. Senzory často obsahují více přijímačů a vysílačů paprsku, které navíc rotují nebo se nějakým způsobem pohybují a tím pokrývají ještě větší prostor.

Výsledná data se vyznačují vysokou přesností (odchylky v jednotkách cm) a společně s daty z dalších senzorů se využívají pro tvorbu map. Vysoká přesnost také s sebou přináší nevýhodu, pokud například sněží nebo prší tak ve výsledném mračnu budou s největší pravděpodobností artefakty. Lidarové senzory mají možnost vidět „skrze stromy“ a to díky velkému množství vyslaných bodů, jednoduše se pár paprsků dostane i do malých děr mezi listy. Druhým důvodem je možnost senzorů detekovat více odrazů jednoho vyslaného paprsku.

2.2 Důležité pojmy

Než přejdeme k samotnému popisu problému a řešení *SLAM*, je třeba si nadefinovat některé základní pojmy, které se budou skrze celou diplomovou práci vyskytovat. U těchto pojmů pro pochopení této práce není důležité znát jejich podrobnosti, jako spíše mít představu o jejich konceptu. Vyjmenované pojmy mají význam i samy o sobě, ale všechny souvisejí nebo se využívají v řešení *SLAM* problémů.

Odometrie

Pojem odometr nejspíše každý vlastník auta zná, používá se pro měření celkové ujeté vzdálenosti auta pomocí měření otáček kol. Odometrie s tímto významem souvisí. Je to proces [6], který umí určit polohu robota při pohybu. K tomuto určení využívá buď naměřená data, jako je například zmíněný počet otáček kol nebo příkazy určené robotovi. Takové příkazy můžou být různého

druhu a přesnosti, např. „jeď 10 metrů dopředu“ ale i „přidej rychlost motorům“. Podstatné je, aby se tyto příkazy daly nějakým způsobem převést na určení nové polohy. V literatuře tento pojem většinou nepopisuje samotný proces, jako spíše konkrétní výsledky, ale dokonce i vstupní data, což jak bylo zmíněno, jsou příkazy nebo naměřená data z robota.

Scan-matching

Pokud robot získal mračno v jednom místě a pohnul se jen o kousek, tak velmi pravděpodobně bude druhé mračno hodně podobné, jen budou hodnoty souřadnic lehce posunuté. Scan-matching nám umožní najít relativní transformaci mezi těmito dvěma mračny. Tento proces je obecný a může být uplatněn i na jiných datových typech než mračnech bodů. Jeden z nejpoužívanějších a nejznámějších algoritmů je *ICP* [7]. Další informace lze najít třeba zde [8]. Scan-matchingu se také říká registrace[3].

Loop closure

Řekněme-li, že mapujeme prostor okolo kruhového objezdu, určování polohy nebude přesné a bude při každém posunutí ujíždět lehce doprava. Chyba se bude neustále akumulovat a nejspíše, když objedeme celý objezd, tak v naší výsledné mapě nebude na sebe navazovat konec a začátek. Využijeme znalosti, že by měly na sebe navazovat, protože se jedná o stejná místa. K takovému zjištění můžeme použít výše uvedený scan-matching a následně mapu opravit. Takové situaci se říká loop closure.

IMU

Je zkratka pro *Inertial Measurement Unit*. Značí se tím systém obsahující akcelerometr a gyroskop. Díky těmto senzorům dokáží produkovat natočení, polohu, rychlost [9]. *IMU* v kombinaci s 2D lidarem můžou tvořit systém, který umí produkovat 3D mračno bodů. Může se jednat o senzor připevněný k rotující části robota a s každou orientací vytvořený 2D sken vložíme do prostoru.

2.3 SLAM - Simultaneous Localisation and Mapping

Veškeré problémy mapování a lokalizace provází nepřesnosti a šum měření, jak při lokalizaci robota ve známé mapě, tak při tvoření mapy se známou pózou robota. Proto se zvlášť lokalizace, mapování i *SLAM* řeší skrze pravděpodobnosti, které interpretují data s nějakou neurčitostí. Pokud známe přesnou pózu, můžeme měření ze senzoru umístit na správnou polohu do mapy. Naopak pokud známe mapu, můžeme, například pomocí scan-matchingu, zjistit přesnou pózu v mapě. *SLAM* řeší problém, který nemá ani mapu ani pózu, tím pádem se musí řešit obojí

[3] `http://pointclouds.org/documentation/tutorials/registration_api.php`

zároveň, díky čemuž je náročnějším problémem než pouhé mapování při známé póze nebo lokalizace při známé mapě. Řešení problému *SLAM* se dělí na dva typy: tzv. *online SLAM* a *offline SLAM*. *Online SLAM* by se dal nazvat jako „*SLAM* za jízdy“, jehož cílem je totiž zjistit pózu a mapu v jedné (aktuální) chvíli. Oproti tomu cílem *offline SLAMu* je zjistit celou trajektorii cesty. Pro detailnější informace nejenom o *SLAMu* doporučuji skvělý kurz dostupný online [10] z Univerzity Freiburgu, knihu *Probabilistic Robotics* [11] což je také jeden z materiálů, který je doporučován v dříve zmíněném kurzu a následně dizertační práci od Joan Solà [12].

Definice

SLAM řeší pomocí pravděpodobností a má svoji definici [11]. Pro *online SLAM* to je:

$$p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}) \tag{1}$$

Index $_{1:t}$ značí, že proměnná má více hodnot zachycených od začátku jízdy robota ($\boldsymbol{z}_{1:t} = \boldsymbol{z}_1, \boldsymbol{z}_2, \cdots, \boldsymbol{z}_t$) a význam proměnných je:

- $\boldsymbol{x}$ reprezentuje stav robota – jeho póza,
- $\boldsymbol{m}$ značí mapu, která je buď reprezentována polohou význačných bodů nebo nějakou prostorovou reprezentací,
- $\boldsymbol{z}$ jsou měření senzoru, může to být mračno bodů, vzdálenost povrchu, ale i obrázek,
- $\boldsymbol{u}$ značí vstupní data odometrie čili třeba příkazy robotovi (dále budou označovány pouze jako příkazy).

Mapu předpokládáme, že máme pořád stejnou v jakémkoli čase, proto nemá index. Tato pravděpodobnostní funkce nám vlastně popisuje jaké jsou hodnoty aktuálního stavu robota a mapy na základě všech předchozích hodnot měření a příkazů robotovi. Řešení *SLAMu*, využívají tzv. *Markovův předpoklad*, který nám říká, že předpoklad o nových hodnotách závisí pouze na předchozím stavu hodnot, ne na všech stavech v minulosti. Tento předpoklad poté umožní přepsat *online SLAM* na

$$p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{x}_{t-1}, \boldsymbol{z}_t, \boldsymbol{u}_t), \tag{2}$$

což se může zdát bezvýznamné, nicméně to umožní využít rekurzivní výpočet nového stavu, na základě pouze předchozího stavu, aktuálního měření a příkazu. Takový předpoklad umožní menší reprezentaci v paměti (máme jen jednu aktuální pózu), ale hlavně tento princip umožní využít rekurzivní algoritmy jako je *Kalmanův filtr* a *filtr částic*. Pro doplnění zde je definice *offline SLAMu*:

$$p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}) \tag{3}$$

Rozdíl je, že místo jednoho stavu, chceme vědět stav po celé trase. Řešení *offline SLAMu* se dá získat kombinací všech řešení *online SLAMu*. Následně pokud v této práci zmíním *SLAM*, tak se bude jednat o *online SLAM*, nebude-li řečeno jinak.

2.3.1 SLAM založený na Kalmanově filtru

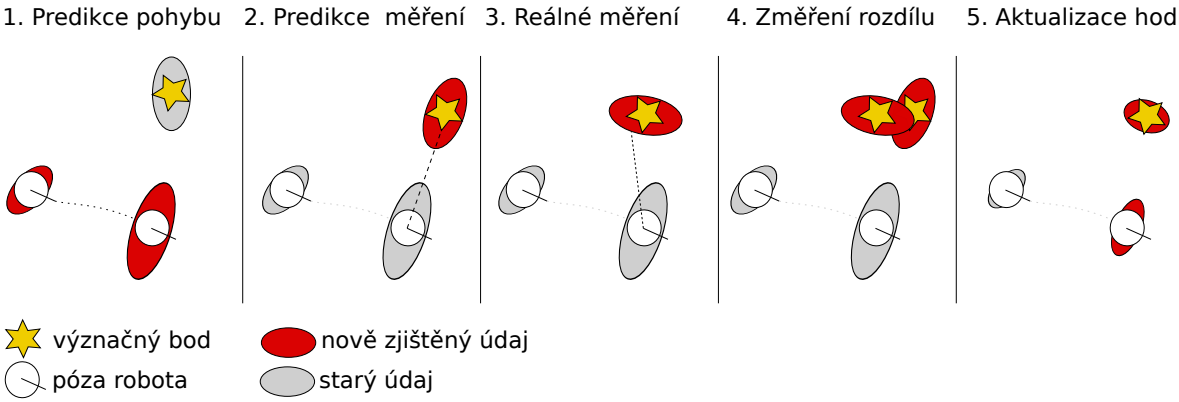
Jedná se o historicky první řešení problému *SLAM* [13]. Hlavní myšlenka je použití nějakou variantu *Kalmanova filtru*, který umí odhadovat následující stav systému (póza a mapa) na základě aktuálního měření a předchozího stavu. *Kalmanovy filtry* mají předpoklad, že řešené hodnoty, musí být v normálním rozdělení. V této rodinně algoritmů je mapa prostředí reprezentována polohou význačných bodů. Díky takové reprezentaci tyto algoritmy nemají vysoké nároky na paměť, ale musí obsahovat dobrý mechanismus na rozeznání význačných bodů z výstupu senzorů. V principu, *SLAM* založený na *Kalmanově filtru* se skládá ze dvou kroků:

1. *predikční*, jehož cílem je určit pravděpodobnost nové pózy robota na základě jeho pohybu (aktualizace pózy pomocí odometrie),

2. *korekční*, který na základě pozorování (výstupu senzorů a rozpoznání význačných bodů) opraví stav robota. Tento krok bere v potaz rozdíl mezi hodnotou očekávanou (vypočítanou na základě predikce pohybu) a mezi hodnotou naměřenou.

Do této kategorie také zapadají algoritmy založené na tzv. *informačním filtru*, kde se využívá velmi podobných principů, jen s jinou pravděpodobnostní reprezentací. Díky tomu jsou rychlejší v korekčním kroku, ale zase pomalejší v kroku predikčním. Příklady řešení založené na x*Kalmanově filtru* jsou *EKF* (*Extended Kalman Filter*) [13] a *UKF* (*Unscented Kalman Filter*) [14], na informačním filtru poté *EIF* (*Extended Information Filter*) [15] a *SEIF* (*Sparse Extended Information Filter*) [16].

Na obrázku č. 2 je znázorněna ilustrace kroků algoritmu řešící problém SLAM založený na *Kalmanově filtru*. Když se robot pohne, předpovíme novou pózu pomocí odometrie, nicméně ta není přesná, je v ní nějaká nejistota, kterou vyjadřuje elipsa okolo robota. Následně vypočítáme jaké měření bychom měli očekávat při předpovězeném pohybu a porovnáme jej s reálným měřením, které má taky svoji nepřesnost. Tyto údaje zkombinujeme a díky toho zvýšíme přesnost nové pózy, polohu význačného bodu, ale i starou pózu, protože tyto hodnoty jsou mezi sebou korelované.



Obrázek 2: Ukázka fungování SLAM algoritmu založeném na Kalmanově filtru, převzato z [17]

V následujících podsekcích je zjednodušeně demonstrována implementace *EKF SLAMu*, který je první algoritmus řešení problému *SLAM* a protože je také nejrozšířenější z rodiny *Kalmanových filtrů*. Jedná se spíše o podsekce pro zájemce, které zajímá, co se vlastně děje uvnitř takové implementace a jaké to má odůvodnění.

EKF SLAM

Jedná se o řešení problému *SLAM* založené na *rozšířeném Kalmanově filtru* (*Extended Kalman Filter*). V *EKF SLAMu* [13] mapa společně s pózou robota je reprezentována náhodným vektorem $\boldsymbol{x}$ v normálním rozdělení. $\boldsymbol{R}$ značí pózu robota, čili jeho polohu $\boldsymbol{p}$ a orientaci $\boldsymbol{q}$, $\boldsymbol{M}$ označuje mapu, n značí počet význačných bodů (označované jako *landmarks*, proto zkratka $\boldsymbol{l}$).

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{M} \end{bmatrix} = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{q} \\ \boldsymbol{l_1} \\ \dots \\ \boldsymbol{l_n} \end{bmatrix} \tag{4}$$

V tomto zjednodušeném vysvětlení je předpoklad, že počet význačných bodů je pořád stejný a že je máme již inicializované. Protože náhodný vektor je modelován normálním rozdělením, tak jsou hodnoty v implementaci uloženy vektorem středních hodnot $\hat{\boldsymbol{x}}$ a kovarianční maticí $\boldsymbol{\Sigma_x}$. Cílem *EKF SLAMu*, je tyto hodnoty udržovat aktuální.

$$\hat{\boldsymbol{x}} = \begin{bmatrix} \hat{\boldsymbol{R}} \\ \hat{\boldsymbol{M}} \end{bmatrix} \qquad \Sigma_x = \begin{bmatrix} \boldsymbol{\Sigma_{RR}} & \boldsymbol{\Sigma_{RM}} \\ \boldsymbol{\Sigma_{MR}} & \boldsymbol{\Sigma_{MM}} \end{bmatrix} = \begin{bmatrix} \Sigma_{RR} & \Sigma_{RL_1} & \cdots & \Sigma_{RL_n} \\ \Sigma_{L_1R} & \Sigma_{L_1L_1} & \cdots & \Sigma_{L_1L_n} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{L_NR} & \Sigma_{L_nL_1} & \cdots & \Sigma_{L_nL_n} \end{bmatrix} \tag{5}$$

Predikční krok v EKF SLAM

Predikční krok odhadne novou polohu robota, na základě jeho pohybového modelu (odometrie). Pro aktualizaci máme funkci g, která umí vypočítat novou pózu robota na základě předchozí pózy a aktuálního příkazu. Příklad, co taková funkce provádí: vstupní příkaz je posunutí o 10 metrů, výsledná nová póza přičte 10 metrů ke staré poloze v závislosti na natočení. Protože *EKF SLAM* reprezentuje stav systému pomocí průměru a kovariance, tak musíme aktualizovat právě je. Proměnná $\boldsymbol{u}$ reprezentuje vektor příkazů.

$$\begin{aligned} \hat{\boldsymbol{x}} &= g(\hat{\boldsymbol{x}}, \boldsymbol{u}) \\ \boldsymbol{\Sigma_X} &= \boldsymbol{F_x} \boldsymbol{\Sigma_x} \boldsymbol{F_x}^\top + \boldsymbol{R} \end{aligned} \tag{6}$$

Kovariance je výše napsaným způsobem aktualizována pomocí transformační matice $\boldsymbol{F_X}$ a je do ní zakomponována nepřesnost reprezentovaná kovarianční maticí šumu $\boldsymbol{R}$. Transformační matice $\boldsymbol{F_X}$ se uvádí jako matice prvních derivací (Jacobiho matice $\boldsymbol{F_X} = \frac{\partial f}{\partial \boldsymbol{x}}$). Intuitivně se dá chápat, že na kovarianční matici $\boldsymbol{\Sigma_x}$ aplikuje transformaci stejným způsobem, jakým funkce g mění hodnoty $\hat{\boldsymbol{x}}$. Pokud se podíváme, jak *EKF SLAM* reprezentuje stav, můžeme si povšimnout, že mapa význačných bodů zůstává stejná, mění se jen poloha a orientace robota. Díky toho můžeme ušetřit na počítání transformace celého stavu systému, resp. jeho vektoru a matic. V níže uvedené matici, jsou vyznačeny hodnoty, které se v predikčním kroku mění. Podle toho se provedou úpravy, aby se zbytečně nepočítaly hodnoty, které se nemění.

$$\begin{bmatrix} \Sigma_{RR} & \Sigma_{RL_1} & \cdots & \Sigma_{RL_n} \\ \Sigma_{L_1R} & \Sigma_{L_1L_1} & \cdots & \Sigma_{L_1L_n} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{L_NR} & \Sigma_{L_nL_1} & \cdots & \Sigma_{L_nL_n} \end{bmatrix} \tag{7}$$

Korekční krok v EKF SLAM

Jak jsem již naznačil, korekční krok v *EKF SLAMu* znamená opravit stav na základě pozorování. Z predikčního kroku máme vypočítanou předpověď stavu $\hat{\boldsymbol{x}}$ a $\boldsymbol{\Sigma_x}$, v tomto kroku máme funkci pozorování h: $\boldsymbol{y} = h(\boldsymbol{x}) + \boldsymbol{v}$, která vrací odhad naměřené hodnoty $\boldsymbol{y}$ (polohu význačného bodu), kde $\boldsymbol{v}$ je šum měření. Funkci h použijeme, abychom mohli vypočítat rozdíl $\boldsymbol{z}$ mezi skutečným měřením a mezi odhadnutým měřením a tím zjistit „důvěryhodnost měření“ a o kolik

potřebujeme odhadnutý stav opravit. Pro predikční krok je třeba vypočítat:

$$\begin{aligned} \hat{\boldsymbol{z}} &= \boldsymbol{y} - h(\hat{\boldsymbol{x}}) \\ \boldsymbol{Z} &= \boldsymbol{H_x} \boldsymbol{\Sigma_X} \boldsymbol{H_x}^\top + \boldsymbol{R} \\ \boldsymbol{K} &= \boldsymbol{\Sigma_X} \boldsymbol{H_x}^\top \boldsymbol{Z}^{-1} \\ \hat{\boldsymbol{x}} &= \boldsymbol{x} + \boldsymbol{K}\hat{\boldsymbol{z}} \\ \boldsymbol{\Sigma_X} &= \boldsymbol{\Sigma_X} - \boldsymbol{K}\boldsymbol{Z}\boldsymbol{K}^\top \end{aligned} \tag{8}$$

První dvě rovnice počítají rozdíl mezi předpokládaným měřením a reálným měřením, kde $\boldsymbol{H_x} = \frac{\partial h}{\partial \boldsymbol{x}}$ je kovarianční matice měření (jak se mění hodnoty výsledné hodnoty z funkce h v závislosti na $\boldsymbol{x}$) a $\boldsymbol{R}$ je kovarianční matice šumu měření. $\boldsymbol{K}$ je tzv. *Kalman gain*, který říká, jak moc se změní stav celého systému na základě měření. Když variabilita předpovězeného stavu je velká (tím pádem nepřesná, to značí velké hodnoty v kovarianční matici $\boldsymbol{\Sigma_X}$) tak *Kalman gain* bude větší, aby se nám stav systému zpřesnil. Naopak, když kovarianční matice $\boldsymbol{Z}$ bude mít velké hodnoty, znamená to, že hodnoty senzoru jsou hodně nepřesné a nechceme, aby toto měření mělo velký vliv na stav systému. Poté co máme vypočítaný *Kalman gain*, změny započteme do stavu systému (tzn. do průměru a kovariance). Tento krok se provádí pro každé pozorování význačného bodu. Taková korekce ovlivní jen hodnoty spjaté s tímto jedním význačným bodem a s pózou robota. Takže opět jako v predikčním kroku je provedena optimalizace výpočtu, nicméně princip platí stejně.

2.3.2 SLAM založený na filtru částic

Princip filtru částic je založen na aproximaci pravděpodobnostního rozdělení množstvím vzorků a váh. Pokud by bylo vzorků teoreticky nekonečno, tak budeme mít přesně požadovanou pravděpodobnostní funkci [18]. Touto reprezentací se zbavíme předpokladu o rozdělení, protože vážené vzorky mohou popisovat jakékoliv rozdělení. Vzorky umožňují reprezentovat i multimodální rozdělení, což se může hodit v případě, že vyjedeme z místnosti a přijedeme do místnosti vedle, která vypadá stejně jako ta, ze které jsme vyjeli (nebo v případě podlouhlé chodby). V principu *filtr částic* funguje, tak že se inicializují hodnoty vzorků na náhodnou hodnotu a poté jedna aktualizace vzorku se skládá z:

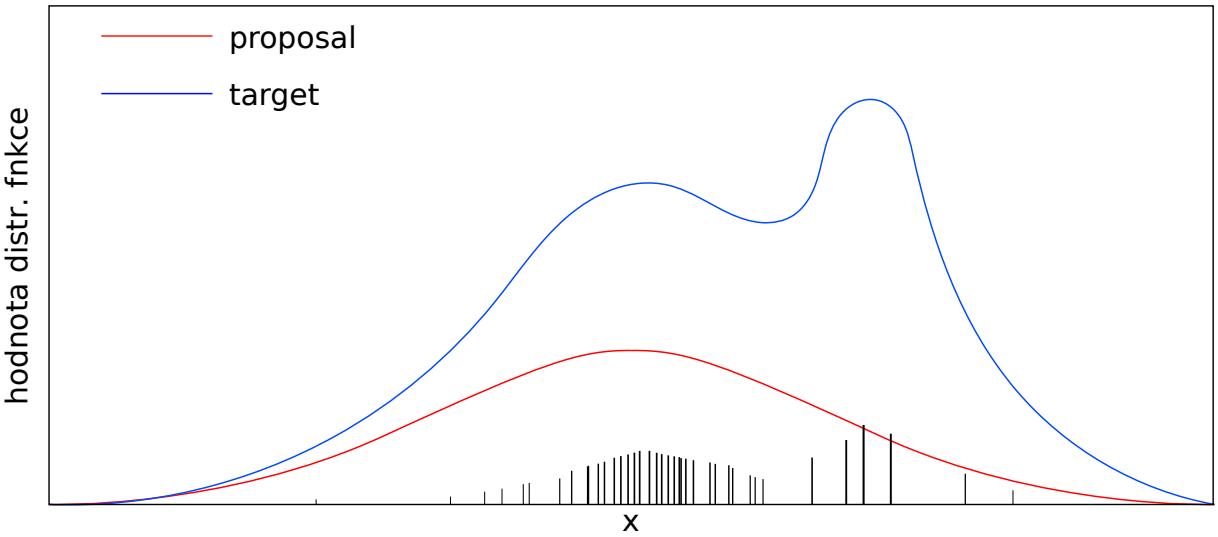
1. aktualizace hodnot vzorku,
2. určení váhy vzorku,
3. převzorkování, což je proces odebrání vzorků s malou váhou a ponechání vzorků s vysokou vahou.

Pokud nemůžeme určit váhu vzorku přímo z rozdělení které počítáme, můžeme využít jiné pravděpodobnostní rozdělení (nazývané *proposal*). Váhu vzorku určíme poměrem mezi požadovaným

(*target*) a *proposal* rozdělením, přehledněji v tomto vzorci, kde x je hodnota vzorku:

$$weight = \frac{target(x)}{proposal(x)} \tag{9}$$

Tomuto principu se říká *Importance Sampling* a na obrázku č. 3 je ukázka tohoto principu. V něm prvně vzorkujeme červenou funkci, proto je počet vzorků v okolí vrcholu červené funkce větší a chceme aproximovat hodnoty z modré funkce. Váhu vzorků značí jejich výška. Obrázek je ilustrační a reálně musí být obsah plochy pod distribuční funkcí vždy roven 1.



Obrázek 3: Ukázka principu Importance Sampling

Na první pohled se může zdát, že opět potřebujeme určit hodnotu z cílového rozdělení, ale tento vzorec se dá pomocí pravděpodobnostních pravidel a předpokladů vyjádřit jinou pravděpodobností, která spočítat jde.

Jako první s použitím principu filtru částic na problém *SLAM* přišel algoritmus *FastSLAM* (*Factored Solution to the Simultaneous Localization and Mapping Problem*) [19], který se dočkal i jeho druhé vylepšené verze [20]. Na něm demonstruji princip použití částicového filtru pro *SLAM*.

FastSLAM

Filtry částic jsou efektivní, pokud náš problém je nízko dimenzionální, protože čím více dimenzí problém má, tím více vzorků potřebujeme pro pokrytí takové vícedimenzionální distribuční funkce. Ve *SLAMu*, který má mapu založenou na význačných bodech, tak náš stavový vektor bude obsahovat všechny polohy význačných bodů a pózu robota. Takový vektor bude nejspíše vysoké dimenze, v závislosti na počtu sledovaných význačných bodů. Proto se v řešení *FastSLAMu* přistupuje k problému, že pokud bychom znali pózu robota, tak mapování je snadný

úkol[4]. K tomu *FastSLAM* využívá tzv. *Rao-Blackwellized particle filter*, který nám umožní od sebe oddělit mapu a pózu robota (vysvětlení proměnných v sekci 2.3):

$$p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_t, \boldsymbol{u}_t) = p(\boldsymbol{x}_t \mid \boldsymbol{z}_t, \boldsymbol{u}_t) p(\boldsymbol{m} \mid \boldsymbol{x}_t, \boldsymbol{z}_t) \tag{10}$$

Protože mapa v tomto případě je tvořená jednotlivými význačnými body a ty jsou na sobě nezávislé, můžeme psát, že celá mapa je výsledkem součinů jejich pravděpodobností

$$p(\boldsymbol{x}_t \mid \boldsymbol{z}_t, \boldsymbol{u}_t) p(\boldsymbol{m} \mid \boldsymbol{x}_t, \boldsymbol{z}_t) = p(\boldsymbol{x}_t \mid \boldsymbol{z}_t, \boldsymbol{u}_t) \prod_{i=0}^{n} (\boldsymbol{l}_i \mid \boldsymbol{x}_t, \boldsymbol{z}_t) \tag{11}$$

Ve výše uvedené rovnici, levý činitel je zvlášť řešení lokalizace na základě měření a příkazů, zatímco v pravé části je n součinů poloh význačných bodů. Lokalizace je prováděna *filtrem částic* a každý význačný bod je řešen dvourozměrným (pro 2D) *rozšířeným Kalmanovým filtrem*. Díky tohoto řešení zvlášť, nemáme jako v *EKF SLAMu* jednu velkou matici stavu, ale více menších. Toto se na první pohled může zdát nevýhodné, ale *EKF* je velmi efektivní na malých dimenzích a navíc při zakomponování měření jednoho význačného bodu se nemusí aktualizovat všechny *EKF*, stačí jen jeden (pozn. i v *EKF SLAMu* se po optimalizaci se nemusí aktualizovat celý stav). Stav uvnitř *FastSLAMu* je reprezentován množstvím vážených vzorků, kde každý vzorek je reprezentován vektorem:

$$\begin{bmatrix} \boldsymbol{p} & \boldsymbol{q} & \boldsymbol{\mu}_1 & \boldsymbol{\Sigma}_1 & \cdots & \boldsymbol{\mu}_n & \boldsymbol{\Sigma}_n \end{bmatrix} \tag{12}$$

Za povšimnutí stojí, že částice má konkrétní hodnotu pózy robota (poloha $\boldsymbol{p}$ a orientace $\boldsymbol{q}$), místo její reprezentace pravděpodobnostním rozdělením. Polohy význačných bodů se naopak modelují normálním rozdělením, proto můžeme vidět průměr a kovarianci.

Pro každý vzorek se aktualizuje póza robota na základě odometrie a poté se projdou všechny význačné body, které se aktualizují pomocí *EKF*. Váha vzorku se počítá jako normalizovaný součin distribučních funkcí normálního rozdělení, kde se pro průměr a kovarianci bere v potaz rozdíl mezi očekávaným měřením a mezi reálným měřením význačného bodu (proč tomu tak je viz originální článek [19], vychází se z rovnice (9)) pro váhu vzorku.

FastSLAM 2.0 [20] používá stejný princip, jen výpočet nové pózy vzorku zahrnuje v sobě informaci měření, protože se předpokládá, že měřící senzory jsou přesné (třeba lidarový senzor). Díky toho, může algoritmus určit novou pózu přesněji (protože pro odhad stavu je více informací) a následně vzorky mají větší šanci na přežití při převzorkování.

FastSLAM díky jeho oddělení lokalizace a mapování se také používá společně s prostorovou reprezentací mapy. Každý vzorek poté musí udržovat celou strukturu mapy, což je paměťově náročné a proto vznikly postupy pro vylepšení. Jeden z nich je úprava výpočtu nové pózy vzorku,

[4] ve článku [19] doslova píšou, že kdybychom měli věštce, který nám umí lokalizuje robota

která nemá na vstupu jen reálnou odometrii, ale i odometrii získánou pomocí scan-matchingu [21].

2.3.3 SLAM založený na optimalizaci grafu

Jedná se o nejnovější přístup k řešení problému *SLAM*. Oproti předchozím dvěma přístupům, které v podstatě řeší *online SLAM* problém, tyto algoritmy naopak řeší *offline SLAM*. Přístupu založeném na optimalizaci grafu se říká *GraphSLAM*. *GraphSLAM* využívá stejnou myšlenku jako *FastSLAM*, že pokud známe pózu robota, tak mapování je snadné. Tento přístup bude popsán s předpokladem použití prostorové reprezentace mapy. Hlavní myšlenkou je optimalizace grafu póz, nejčastěji pomocí metody nejmenších čtverců. Možná si říkáte, že *SLAM* je přece problém pravděpodobnostní, jak optimalizace grafu souvisí s maximalizací pravděpodobnosti? Dá se totiž dokázat, že minimalizace chyby grafu je maximalizace pravděpodobnosti nezávislých náhodných proměnných v normálním rozdělení (článek [22] a přednáška [23] od jednoho ze spoluautorů článku). Pro lepší pochopení principu *GraphSLAMu* prvně popíšu metodu nejmenších čtverců. Je dobré rozumět principu *GraphSLAMu*, protože na něm je založen *Cartographer*, který se používá v praktické části.

Metoda nejmenších čtverců

Metoda nejmenších čtverců řeší optimalizaci parametrů, které jsou omezené podmínkami [23]. Využívá se, když máme tzv. předeterminovaný systém, což značí, že máme více podmínek, než parametrů. Každá podmínka je vlastně rovnicí, závislá na optimalizovaném parametru. Taková podmínka může být třeba rozdíl předpovězené hodnoty měření (výsledek funkce f) a reálného měření $\boldsymbol{z}$. To nám určuje chybu z jedné rovnice, součet všech chyb tvoří chybu celého řešení. Chyba i-té rovnice se dá vyjádřit jako:

$$\boldsymbol{e}_i(x) = \boldsymbol{z}_i - f_i(\boldsymbol{x}). \tag{13}$$

Cílem této optimalizační metody je najít takové hodnoty parametrů, aby celková suma chyb $\boldsymbol{e}_i$ na druhou, byla co nejmenší. Pokud rozdíl $\boldsymbol{z}_i$ a výsledku funkce f_i je skalár, stačí ho dát na druhou. Jakmile je chyba vyjádřena vektorem, tak z jedné rovnice získáme výslednou hodnotu chyby takto:

$$e_i(x) = \boldsymbol{e}_i^\top(x)\boldsymbol{\Omega}_i\boldsymbol{e}_i(x) \tag{14}$$

V této rovnici nikde na druhou není, proč takový název metody? Druhá mocnina zde pořád je, pokud vynecháme Ω_i a zkusíme si provést násobení předepsané vzorcem, tak zjistíme, že výsledná hodnota chyby bude součet všech prvků vektoru $\boldsymbol{e}_i$ na druhou. Matici Ω_i se říká informační matice (je to invertovaná kovarianční matice) a umožní určit váhu prvkům vektoru chyb $\boldsymbol{e_i}$. To se dá představit, že čím větší kovariance (čili nejistota) dané veličiny, tím menší

hodnota v matici Ω_i a tím méně tato veličina bude přispívat svojí částečnou chybou do celkové chyby.

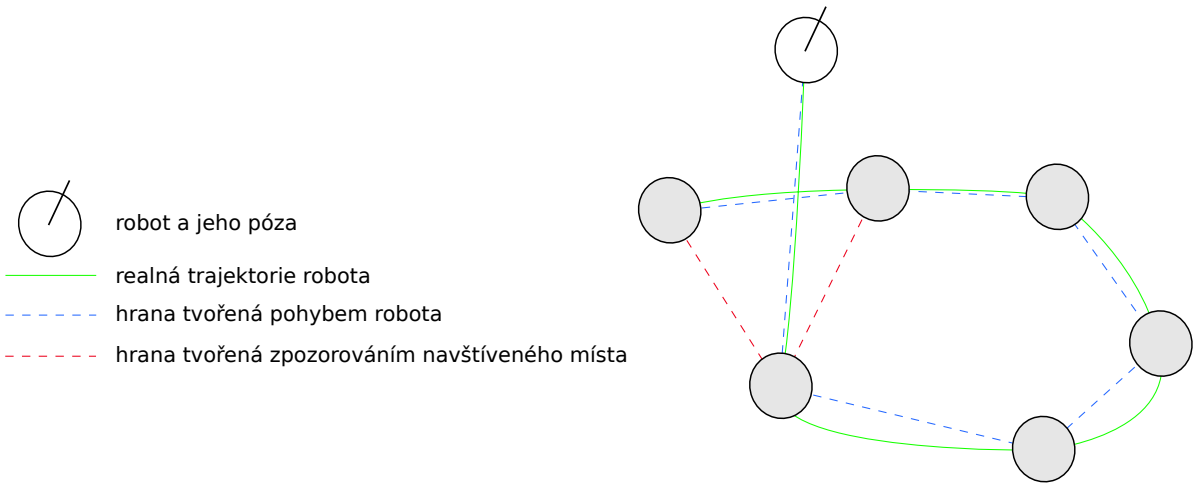
Jedno z řešení nejmenších čtverců je mít odhad správného řešení a poté se iterativně postupným spádem blížit ke globálnímu minimu chybové funkce. Protože chybová funkce je velmi pravděpodobně nelineární, tak to zahrnuje linearizaci v blízkém okolí a nalezení nejmenší hodnoty v tomto okolí, pomocí vyřešení soustavy lineárních rovnic. Buď tento proces iterujeme tak dlouho, až se nám chyba nemění nebo až dosáhneme předem nastaveného počtu iterací. Další poznámka je, že někdy globální minimum může být jinde, než jsme zkonvergovali a musel by iterativní proces přijmout horší hodnotu, než bylo doposud nejlepší řešení.

GraphSLAM

V *GraphSLAMu* [22] si robot co chvíli během své cesty vytvoří nový záznam aktuální pózy a tento záznam se stane uzlem v grafu póz (kterému se také říká trajektorie). Hrany mezi uzly obsahují transformace v homogenních souřadnicích mezi pózami. Hrany vznikají dvěma způsoby:

- pohybem robota (odometrií),
- rozpoznání předem navštíveného místa (k tomuto se dá využít scan-matching), také se těmto hranám říká virtuální odometrie.

Takový graf je ilustrován na obrázku níže.



Obrázek 4: Ukázka grafu pro algoritmus GraphSLAM

V grafu bude vždy $n-1$ hran přímých pohybů, zatímco hran virtuální odometrie může být podstatně více. Pózy uložené v uzlech jsou proměnné, které chceme optimalizovat a hrany tvoří podmínky pro tyto proměnné. *GraphSLAM* se snaží tento graf optimalizovat pomocí metody nejmenších čtverců. Když jsou polohy uzlů v grafu od sebe vzdálené 10 metrů, ale výsledek měření určí vzdálenost 1 metr, tak tyto podmínky vytvoří velkou chybu, která se musí optimalizovat změnou póz v uzlech a proto optimalizace grafu funguje.

GraphSLAM bývá rozdělován na frontend a backend. Frontend se stará o hledání virtuální odometrie a tvorbu grafu, zatímco cílem backendu je optimalizace grafu. Kvalita frontendu je závislá na optimalizaci backendu, protože frontend by neměl hledat předem projetá místa v celém grafu, ale jen okolo jeho aktuální polohy (a staré pózy grafu aktualizuje právě backend). Frontend je často závislý na konkrétním senzoru, u backendu taková silná závislost není, protože pracuje s hotovým grafem. Nyní bude naznačeno, jak se graf optimalizuje.

Každá hrana tvoří podmínku a ta má chybu $\boldsymbol{e}_{ij}$, kde i,j indexy uzlů v grafu, $\boldsymbol{Z}_{ij}$ značí naměřenou transformaci mezi uzly a $\boldsymbol{X}_i$ je póza i-tého uzlu v homogenních souřadnicích, chyba jedné hrany se poté definuje:

$$\boldsymbol{e}_{ij} = vec(\boldsymbol{Z}_{ij}^{-1}(\boldsymbol{X}_i^{-1}\boldsymbol{X}_j)) \tag{15}$$

Argument funkce *vec* se dá chápat, jako rozdíl mezi transformací naměřenou a zaznamenanou, protože pokud bude zaznamenaná transformace $(\boldsymbol{X_i}^{-1}\boldsymbol{X_j}))$ stejná jako naměřená $\boldsymbol{Z}$, bude výsledek součinu nula (a také výsledná chyba bude nulová). Funkce *vec* nám pouze převede matici z homogenních souřadnic do vektoru, který obsahuje translaci a rotaci transformace. Důvodem je, že transformační matice má navíc hodnoty, které se nemůžou nijak měnit a ty optimalizovat nechceme.

Když je definována chybová funkce, tak se dá přímo využít metody nejmenších čtverců k její minimalizaci. Nicméně to by bylo hodně pomalé a proto nastíním efektivnější řešení. V sekci pro seznámení s metodou nejmenších čtverců jsem zmínil, že se iterativně řeší lineární systém. Pro pochopení myšlenky optimalizace není nutné pochopit, jak se k tomuto lineárnímu systému došlo. Tento systém se značí $\Delta\boldsymbol{x} = -\boldsymbol{H}^{-1}\boldsymbol{b}$ jehož výsledkem je inkrement optimalizovaných parametrů, kde hodnoty $\boldsymbol{H}$ a $\boldsymbol{b}$ se získají pomocí:

$$\boldsymbol{b}_{ij} = \sum_{ij} \boldsymbol{J}_{ij}^{\top}\boldsymbol{\Omega}_{ij}\boldsymbol{e}_{ij} \qquad \boldsymbol{H}_{ij} = \sum_{i} \boldsymbol{J}_{ij}^{\top}\boldsymbol{\Omega}_{ij}\boldsymbol{J}_{ij} \tag{16}$$

$$\boldsymbol{b} = \sum_{ij} \boldsymbol{b}_{ij} \qquad \boldsymbol{H} = \sum_{ij} \boldsymbol{H}_{ij} \tag{17}$$

Čili pro každou hranu musíme znát Jakobiho matici prvních derivací $J_{ij} = \frac{\partial \boldsymbol{e}_{ij}(\boldsymbol{x})}{\partial \boldsymbol{x}}$. Protože $\boldsymbol{x}$ jsou pózy v uzlech a chybová funkce $\boldsymbol{e}_{ij}$ závisí pouze na uzlech i a j, tak výsledná matice derivací bude nulová kromě těchto dvou hodnot. Díky tohoto bude matice $\boldsymbol{H}$ řídká, což znamená, že bude obsahovat málo nenulových hodnot, čehož se poté využije k efektivnímu řešení lineární soustavy.

3 Použitá zařízení a technologie

Teoretickou stránku bychom měli za sebou, pojďme se podívat konkrétněji, co všechno jsem k provedení práce potřeboval a použil. Prací jsem se mohl zabývat, protože jsem měl přístup k lidarovému senzoru Velodyne VLP-16, který byl později namontován na zmenšený model auta. Cílem této práce nebylo *SLAM* implementovat, ale využít již hotové open-source řešení. Tímto vybraným řešením je *Google Cartographer*, jak funguje a proč jsem jej právě zvolil bude popsáno níže *Google Cartographer* je snazší používat dohromady s Robot Operating System (dále jen *ROS*), proto jej v práci využívám také. *ROS* navíc usnadňuje používání samotného VLP-16. V této sekci se dozvíme, co vlastně ten *ROS* je a jak usnadňuje práci vývojářům nejen robotů, na konkrétním příkladu užití.

3.1 Velodyne VLP-16

Celým jménem Velodyne Puck™ VLP-16 se jedná o nejmenší lidarový senzor od Velodyne. Je to zhruba 7 cm vysoký válec v průměru zhruba 10 cm. Má 16 dvojic přijímačů a vysílačů paprsku, které rotují uvnitř a tím vytvářejí zorné pole 360° horizontálně a 30° vertikálně (15° nahoru a 15° dolů). Měří až do vzdálenosti 100m s přesností +/- 3 cm a produkuje až 0.3 milión bodů za sekundu. V sekci o lidaru 2.1 jsem zmínil, že lidar má možnost „vidět přes stromy“. VLP-16 nabízí takovou možnost s jeho třemi možnostmi nastavení return módu. Ten určuje, jestli půjde na výstup ten nejsilnější odraz paprsku (return mód strongest), ten poslední (mód last) a nebo oba (mód dual).

Obrázek 5: Velodyne Puck™ VLP-16 (převzato z [5])

Senzor nabízí konfigurační webové rozhraní, které je ve výchozím nastavení dostupné na adrese
`192.168.1.201`. V něm se dá nastavit počet rotací za minutu, return mód, velikost zorného pole, datové porty a dokonce se může stát ze senzoru i *DHCP* server.

Data posílá VLP-16 skrze Ethernet pomocí *UDP* globálního broadcastu. Pakety mají proprietární formát a obsahují mj. vzdálenosti bodů, 3D pozici uloženou v rotačních úhlech a synchronizovanou časovou značku. K VLP-16 se dá připojit GPS, podle kterého si senzor dokáže synchronizovat čas. Pokud GPS není připojeno, tak při startu senzoru se nastaví interní časovač na nulu. Tyto informace jsem vyčetl z dodávané dokumentace v balení, která je dostupná i online

z [5]. V této práci jsem pro přijímání dat ze senzoru použil již implementovanou knihovnu (více v 4.1)

3.2 Robot Operating System

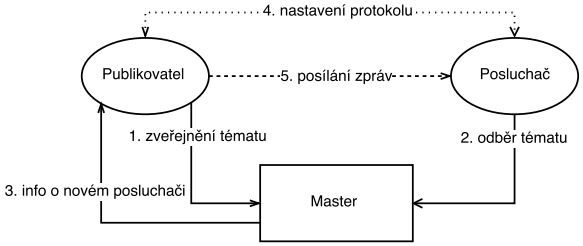
Ačkoliv je v názvu operační systém, nejedná se o systém jako je třeba Microsoft Windows. Je to framework, který poskytuje vývojářům robotů (ale nejen jim) jednotné prostředí, skrze které můžou oddělené komponenty komunikovat mezi sebou. Dále *ROS* nabízí kolekci nástrojů, programů a knihoven pro zjednodušení vývoje komplexního softwaru, který je spojený z více částí. Tyto části tvoří balíčky, které jsou všechny open-source a doporučuje se, aby funkcionalita balíčku byla použitelná i samostatně bez *ROSu*. Balíčky můžou být napsány v různých jazycích (nejčastěji C++ a Python) a dokonce ani nemusí být na stejném zařízení. Celý *ROS* je zdarma a open-source [24]. Jmenovitě *ROS* obsahuje:

- unifikované předávání zpráv a volání služeb,
- jednotný systém pro nahrání, ukládání a přehrávání dat skrze tzv. *bags*,
- server na ukládání a zjišťování parametrů,
- nástroje pro vizualizaci,
- nástroje pro tvoření softwaru pro ROS,
- konzolové nástroje pro ladění a testování,
- balíčky pro „typické“ problémy, jako je geometrie, transformace mezi souřadnicovými systémy, definované zprávy pro často používané typy jako je mračno bodů, vektory, body, obrázek, teplota a další [25].

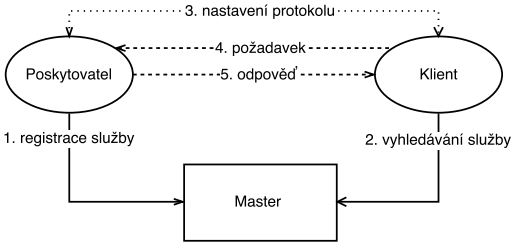
Než krátce popíšu koncepty *ROSu* ukážu je na konkrétním příkladu. Řekněme-li, že máme mini počítač s kamerou a chtěli bychom obrázky analyzovat a zjistit počet volných míst na parkovišti. Náš mini počítač na to nemá dost výkonu, tak chceme provést výpočet na jiném počítači. Dejme tomu, že jsou propojené Ethernetem. *ROS* nabízí všechno co je potřeba. V mini počítači spustíme ovladač, který bude data získávat z kamery. Na druhém počítači, spustíme výpočetní program, který bude přijímat obrázky. Výhoda *ROSu* je v tom, že teď můžeme kdykoliv vyměnit zdroj obrázků např. za soubor, jiný minipočítač, lokální kameru, dokonce můžeme představit před zdroj i filtr a žádný kód se ve výpočetním uzlu nezmění. Ten dokonce ani nemusí mít přímo zadanou adresu mini počítače, protože výpočetní program se pouze zeptá master serveru, jestli je u něj někdo zaregistrovaný, kdo publikuje obrázky pod předem domluveným názvem. Master mu poté dá adresu a začne přímá komunikace. Jak ovladač, tak výpočetní program se v *ROSu* nazývají uzly.

3.2.1 Unifikovaný systém zpráv a služeb

Uzly mezi sebou komunikují pomocí zpráv a služeb. Zprávy mají jednotný formát definovaný `.msg` souborem. Můžou obsahovat základní datové typy jako je `int`, `char`, `float`, jejich pole a následně i vnořené jiné zprávy. Kompilací balíčku s definovanými zprávami se vytvoří třídy pro různé programovací jazyky. Tyto zprávy se poté publikují do témat, které mají svůj název a typ. Navázání komunikace pro posílání zpráv můžeme vidět na obrázku č. 6. Jakmile je komunikace navázána, ta nejčastěji probíhá pomocí *TCP* nebo *UDP*. Master poskytuje možnost registrace uzlů, udržuje seznam služeb a zpráv a také poskytuje server parametrů.



Obrázek 6: ROS diagram posílání zpráv



Obrázek 7: ROS diagram volání služby

Posluchač může zprávy pouze přijímat, ale nemůže zveřejňování zpráv nějak ovlivnit. Služby oproti tomu jsou velmi podobné klasickému volání funkcí. Posluchač musí aktivně požádat uzel, který službu poskytuje o její provedení. Služba je definovaná parametry požadavku a parametry odpovědi. Ty můžou mít stejné datové typy jako zprávy. Průběh komunikace při volání služby je na obrázku č. 7.

3.2.2 Uzly – nodes

Uzel je samostatnou výpočetní jednotkou. Uzly jsou na sobě nepřímo závislé, s ostatními uzly komunikují skrze zprávy a služby. O publikovaných zprávách a službách se dozví od mastera a poté uzly můžou navázat přímou komunikaci mezi sebou. Uzly jsou pouze závislé na typu zpráv a služeb, nezáleží na tom, kdo zprávy a služby poskytuje a proto jsou na sobě v podstatě nezávislé [26].

Tato vlastnost je silnou stránkou *ROSu*, protože díky ní můžeme relativně snadno uzly vyměnit a zkoušet různé algoritmy, které řeší stejnou věc, tak jak bylo řečeno v příkladu o parkovacích místech. Také díky této vlastnosti může fungovat jednotný systém pro nahrávání, ukládání a přehrávání zpráv.

3.2.3 Komunikace se HW zařízeními

ROS sám o sobě neobsahuje způsob, jak získávat data z různých senzorů a zařízení. Pro tento účel existují balíčky ovladačů, které dělají to, co se od ovladačů čeká. Dají se v nich nastavovat různé parametry pro konkrétní zařízení, ale hlavně získávají data, které následně převádí do uniformních zpráv. Seznam oficiálně podporovaných je na `http://wiki.ros.org/Sensors`.

3.2.4 Frames – rámce

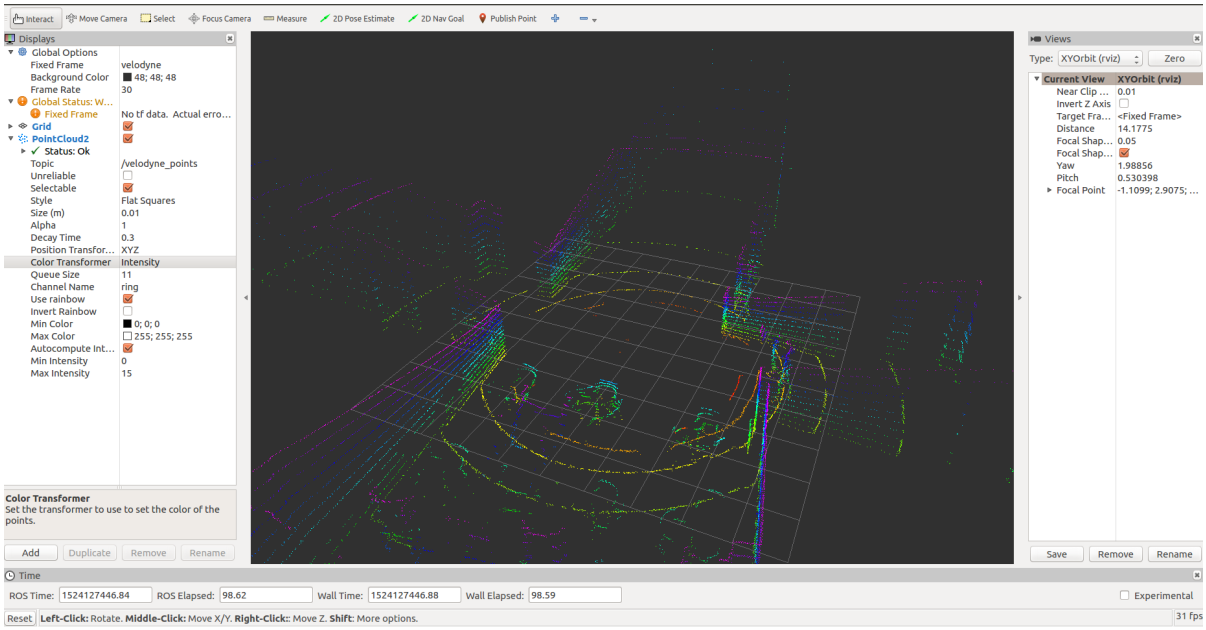
V *ROSu* najdeme pojem rámec často, proto je důležité tento koncept pochopit. Rámec definuje, ke které póze jsou souřadnice v něm definovány. Nejlépe se tento koncept vysvětluje na příkladu. Řekněme-li, že senzor produkuje polohu překážek, které jsou relativní k póze senzoru (třeba bod $(x, y, z) = (1, 0, 0)$ bude značit překážku přímo před senzorem ať už je senzor umístěn jakkoliv). O těchto pozicích poté řekneme, že jsou v rámci senzoru. Pokud polohy překážek chceme zobrazit na mapě, musíme znát pózu senzoru v této mapě a transformovat naměřené polohy překážek, aby byly relativní k počátku mapy.

3.2.5 Vizualizace

ROS v základních instalacích obsahuje 3D vizualizační nástroj *Rviz* a doplňky s grafickým rozhraním *rqt_common_plugins* nebo *rqt_robot_plugins*.

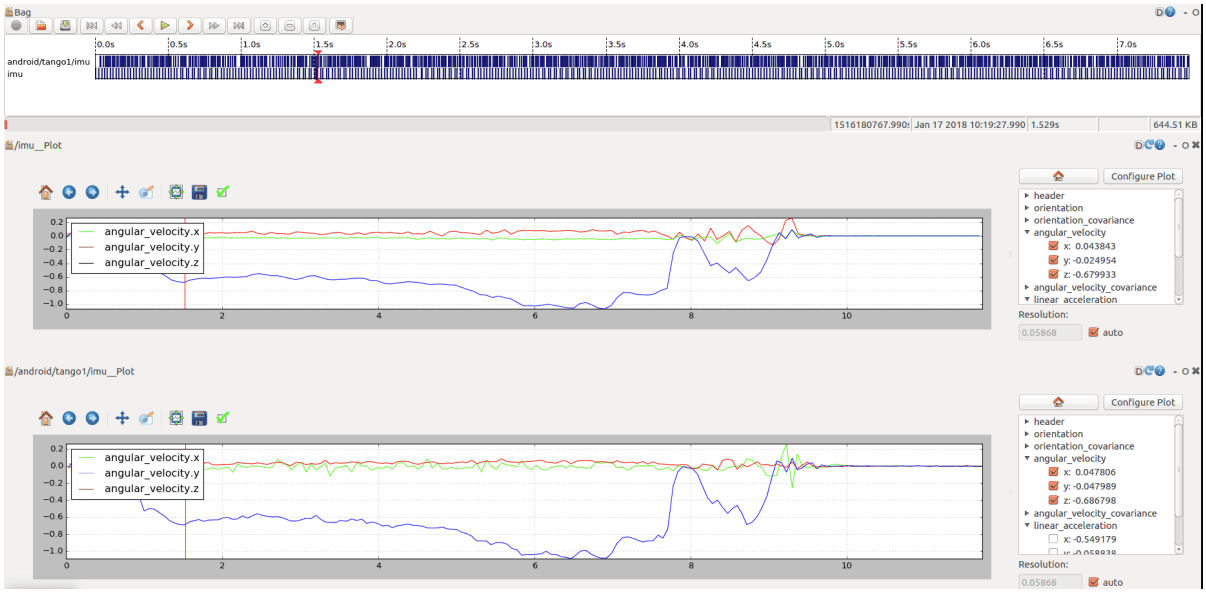
Rviz v základu nabízí vizualizaci základních typů jako je mračno bodů, laserový sken, cesta, jednotlivý bod, mapa, model robota a další[5]. *Rviz* dále také samozřejmě umožňuje napsat si vizualizaci vlastního typu, což je třeba obsahem balíčku *Cartographeru* i *Octomapy* (popis v 3.3 a 3.4). Na stránce `http://wiki.ros.org/rviz/UserGuide` se poté nabízí sice obrázky ze starší verze, ale pořád platné koncepty. Pro základní vizualizaci bych označil ovládání za intuitivní. Jedinou věc, kterou zde zmíním je tzv. *fixed frame*. Použiji zde příklad z předchozí sekce. Řekněme-li, že chceme zobrazit data relativně k senzoru, nastavíme *fixed frame* na rámec senzoru. Pokud chceme vidět polohy překážek v globální mapě, tak nastavíme *fixed frame* na rámec mapy. Příklad takové vizualizace mračna na obrázku č. 8.

[5] `http://wiki.ros.org/rviz/UserGuide#Built-in_Display_Types`



Obrázek 8: Vizualizace 3D mračna pomocí *Rvizu*

Z *rqt_common_plugins* jsem nejvíce používal nástroj *rqt_bag*, který umí vizualizovat průběh a obsah nahraných zpráv ve formátu `bag`. Dále umí zobrazit hodnoty zpráv, kreslit grafy hodnot v čase a přehrávat daný záznam. Ukázku tohoto nástroje můžete vidět na obrázku č. 9.



Obrázek 9: Vizualizace `bag` souboru pomocí *rqt_bag*

3.3 Google Cartographer

ROS obsahuje již implementované *SLAM* algoritmy ze všech rodin přístupů. Například pro *EKF SLAM* balíček `mrpt_slam`[6], pro *FastSLAM* balíček `gmapping`[7] a pro *GraphSLAM* balíček `cartographer`. Pro 3D *SLAM*, který bere jako vstup mračno bodů byl dostupný pouze `cartographer` a `BLAM`[8] oba z rodiny *GraphSLAM*. Vyzkoušel jsem oba, ale protože *Cartographer* měl velmi pozitivní ohlasy, je stále ve vývoji, nabízí relativně dobrou dokumentaci a komunikaci ze strany jeho vývojářů.

Google Cartographer [3] je veřejně dostupný open-source software pro řešení *SLAM* problému jak ve 2D tak ve 3D. Cartographer je napsaný v C++ a využívá obecně používaných knihoven jako je *Boost*, *Eigen* a *Point Cloud Library*. Je vysoce konfigurovatelný a umožňuje zpracovávat data z různých typů senzorů. *Cartographer* je rozdělen na dvě hlavní části:

- *cartographer* – samotný *SLAM* algoritmus,
- *cartographer-ros* – propojení projektu *cartographer* s *ROSem*.

Google tento software používá pro mapování vnitřního prostoru, kde není dostupný signál GPS. Příklad použití je mapování hotelu v San Franciscu, který je dostupný na mapách[9].

3.3.1 Základní popis

Cílem *Cartographeru* není vymyslet nové řešení *SLAM* problému, ale spíše se zaměřit na výkon a efektivitu těch stávajících. Implementuje metodu *GraphSLAM* a rozděluje mapu na menší části tzv. submapy, do kterých se akumuluje konfigurovatelné množství skenů (skenem je myšleno více měření seskupených do jednoho, v našem případě mračno bodů zachycující záběr 360°). Každý sken tvoří uzel v grafu. Lokální scan-matcher se snaží minimalizovat chybu uvnitř submapy. Jakmile je submapa dokončena, do ní vložené skeny jsou používány pro hledání loop closure. Globální scan-matcher poté vyhledává shodu pouze v dostatečně blízkých submapách a tvoří nové hrany v grafu, čili přidává podmínky do optimalizačního problému. Jednou za předem nastavený počet uzlů v grafu se spustí optimalizace grafu pomocí metody nejmenších čtverců.

3.3.2 Přehled systému

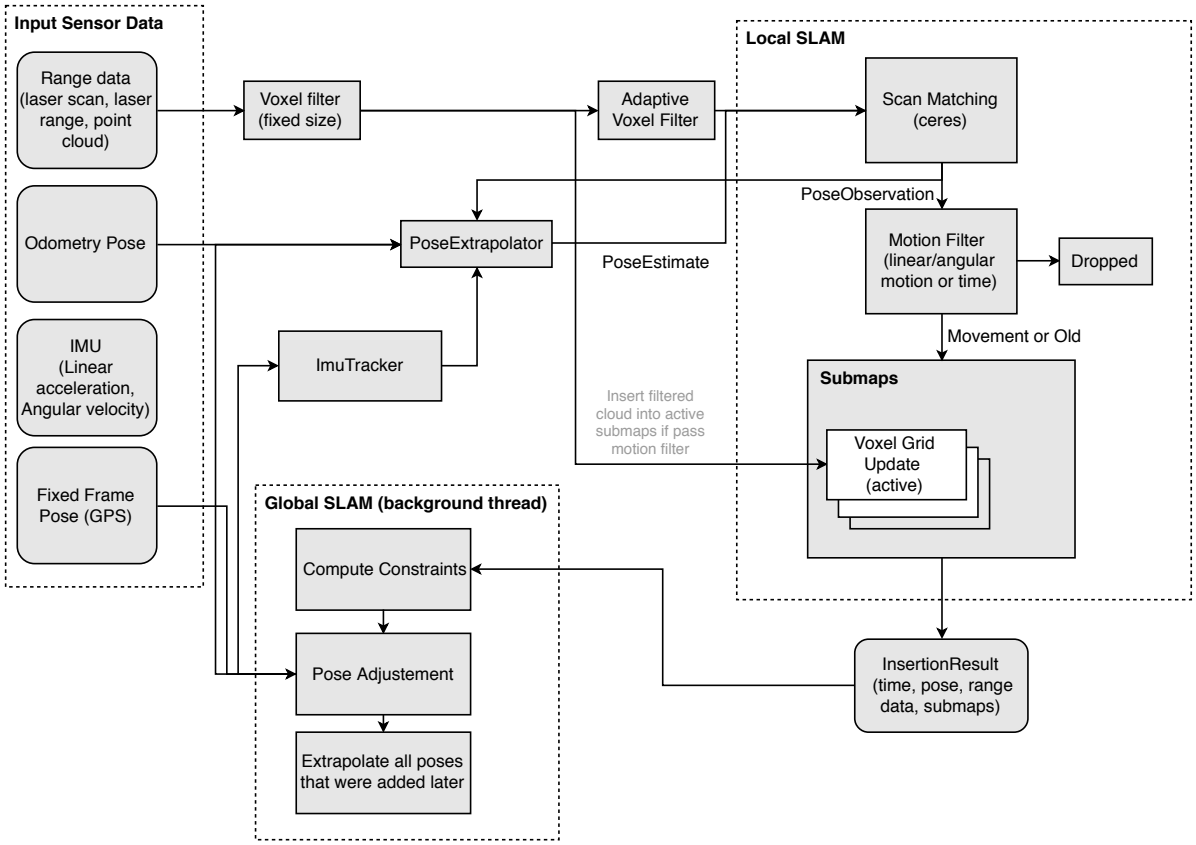
Celý přehled je na obrázku č. 10. Systém během času sbírá data ze čtyřech různých zdrojů. Každé vstupní mračno bodů projde voxel filtrem, jehož cílem je nahradit zašuměné polohy bodů centroidy zarovnanými do mřížky. Jakmile je nasbíráno dostatečný počet dat, vznikne jeden sken. Ten poté projde adaptivním voxel filtrem pro další zmenšení množství dat.

[6] `http://wiki.ros.org/mrpt_slam`

[7] `http://wiki.ros.org/gmapping`

[8] `https://github.com/erik-nelson/blam`

[9] `https://www.google.com/maps/place/San+Francisco+Marriott+Marquis/@37.7854503,-122.4042569,19z/data=!4m5!3m4!1s0x8085808636673e71:0xebc2cf7c5bf2d655!8m2!3d37.7852279!4d-122.404389`



Obrázek 10: Přehled systému Cartographer (převzato z [27])

Lokální SLAM

Vyfiltrované mračno společně s extrapolovanou pózou na základě předchozích dat z *IMU* a z předchozího scan-matchingu jdou na vstup scan-matcheru. Jeho výstupem je upravená póza, která maximalizuje pravděpodobnosti v submapě. Tato póza poté jde skrze filtr pohybu, který určí, jestli není moc blízko pózy staré. Pokud filtr určí, že je moc blízko, použije se stará póza, aby se netvořily zbytečně nové uzly v grafu. Následně se akorát vloží filtrovaná, zarovnaná data do submapy. *Cartographer* nepoužívá pro scan-matching *ICP*, ale metodu nejmenších čtverců.

Globální SLAM

Na pozadí poté běží tzv. globální *SLAM*, jehož cílem je opravit časem akumulovanou chybu lokálního SLAMu. Hledá přídavné hrany do grafu, které tvoří omezení do optimalizačního problému. Tyto omezení pro optimalizační problém jsou hledány pouze v nejbližších submapách.

Periodicky se spouští optimalizace grafu póz. Tato optimalizace bere v potaz omezení, které vzniknou přesunem robota a omezení, které přibudou vyhledáním předem navštívených oblastí. Poté co proběhne optimalizace póz, v grafu se upraví pózy uzlů i submap. Nakonec se projdou všechny pózy v uzlech grafu, které nebyly optimalizovány a extrapolují se i jejich polohy.

3.4 Octomap

Octomap je *ROS* balíček a samostatná knihovna napsaná v C++, která umožňuje reprezentaci 3D světa pomocí voxelové mřížky v prostoru. Pro efektivnější uložení voxelů knihovna používá oktantové stromy [28]. Knihovna je v podstatě hlavní implementace 3D reprezentace světa a používají ji i další frameworky, které tvoří plány pohybu a kontrolují kolize. V implementaci ji sice nepoužívám, ale během vývoje jsem s ní experimentoval.

3.5 Point Cloud Library

Je to hlavní open-source knihovna pro zpracování 3D mračen. Je napsána v C++ a zpracovávání mračen je mnohdy prováděno paralelně za pomocí *OpenMP*, ale také podporuje zpracování na grafických kartách. Je rozdělena do více modulů, kde každý má jiný účel. Podporuje různé typy mračen, jejich filtrování, segmentaci, hledání klíčových bodů, ukládání a načítání dat v obecně podporovaných formátech. Je zde implementovaný neustále zmiňovaný scan-matching i výše uvedený voxel filtr. *PCL* je také použito na pozadí ROSu pro reprezentaci mračen a pro jejich zpracování [29]. V implementační části ji využívám pro kontrolu kolizí.

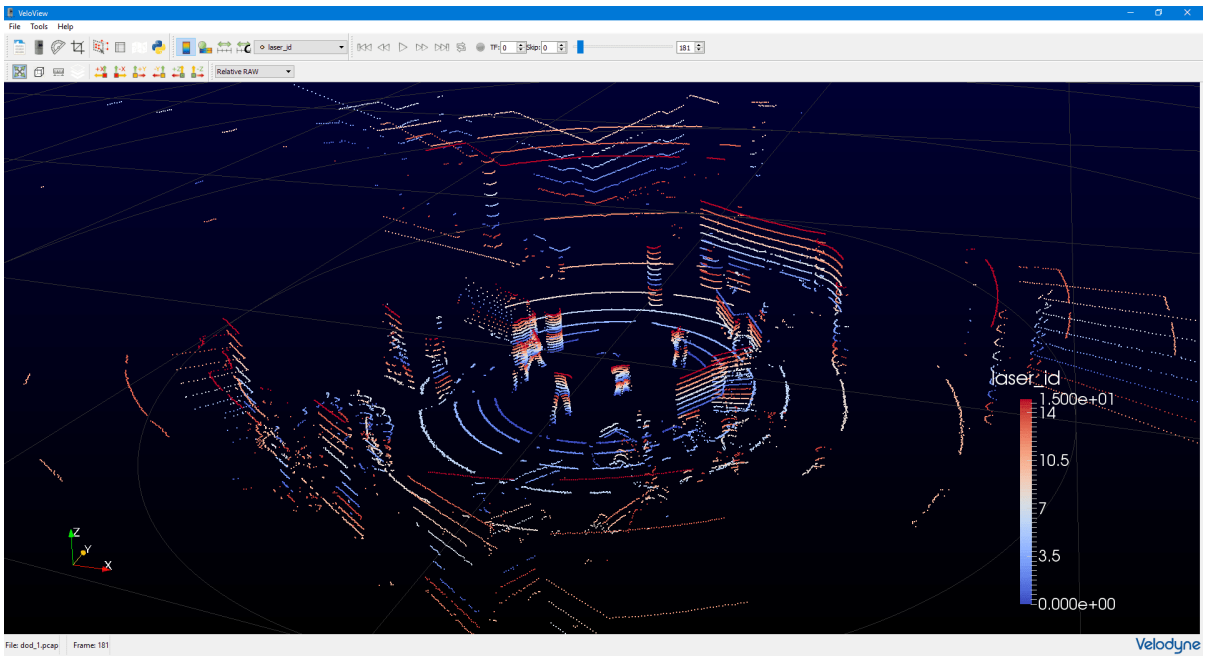
4 Konfigurace a instalace

Když jsme se seznámili s klíčovým softwarem a hardwarem, který je použit v této práci, pojďme se podívat, jak je všechno propojeno dohromady do jednoho funkčního celku. Nejdříve zjistíme jak se pracuje s VLP-16 a jak se propojuje s *ROSem*. Poté se dozvíme o problému, který nastal díky neexistujícímu exportu 3D dat z *Google Cartographeru* a na závěr samotnou konfiguraci *Google Cartographeru*. Všechno co je psáno v této sekci bylo potřeba pro zprovoznění testovacího prostředí.

4.1 Seznámení s VLP-16

Společně s VLP-16 je dodávaný USB flash disk, který obsahuje dokumentaci, návody, ukázková data a v neposlední řadě program *VeloView* (ukázka na obr. 11), kterým se dají přijímaná data zobrazovat. Ukázková data mají různá nastavení senzoru a jsou uloženy ve formátu `pcap`, jinými slovy jde akorát o zachycený datový tok paketů. Program *VeloView* obsahuje možnost zobrazit příchozí data ze senzoru, ale i data uložená ve formátu `pcap`. Tento program je multiplatformní a jelikož jsem prováděl testování na virtuálním stroji, skvěle se mi hodil k zjišťování průchodu dat.

Pro simulaci reálného zařízení, jsem si spustil další virtuální stroj s nainstalovaným programem `tcpreplay`[10], který slouží k přehrávání `pcap` souborů.



Obrázek 11: Ukázka VeloView s daty ze školního senzoru VLP-16 na systému Windows

[10] `http://tcpreplay.synfin.net/wiki/tcpreplay`

Data jsem nicméně potřeboval zpracovat, takže jsem poté vyzkoušel zabudovanou funkcionalitu knihovny *PCL*. K tomu se dá využít třída `VLPGrabber`[11], která taky umí načíst data jak ze souboru `pcap`, tak přímo data posílaná ze senzoru. Nicméně nakonec se ukázalo, že je lepší využít *ROS* a proto bylo třeba najít ovladač k VLP-16.

VLP-16 společně s ROS

První je potřeba mít nainstalovaný na systému samotný *ROS*. Protože jsem chtěl používat *Google Cartographer*, tak jsem si vybral verzi *ROS Kinetic Kame*. Pro instalaci jsem následoval kroky z `http://wiki.ros.org/kinetic/Installation/Ubuntu`.

Pro získávání dat z VLP-16 se dá využít ovladače pro Velodyne zařízení. Ten v normální instalaci chybí, dá se doinstalovat skrze `sudo apt-get install ros-kinetic-velodyne` nebo přímo ze zdrojů na `https://github.com/ros-drivers/velodyne`.

ROS ovladač pro Velodyne zařízení

Tento ovladač obsahuje čtyři balíčky. Pro mě byl nejpodstatnější balíček `velodyne-pointcloud`, protože obsahuje konfiguraci `launch` souboru `VLP16_points.launch`[12]. Tento launch soubor spustí tři uzly, které mezi sebou komunikují. Tyto uzly a jejich úkoly jsou:

- `velodyne_driver` přijímá data vysílaná senzorem buď z *UDP* broadcastu nebo z `pcap` souboru a vnitřně si ho převádí na svůj typ *ROS* zprávy `VelodynePacket`, které potom kombinuje dohromady jako jeden `VelodyneScan`,
- `velodyne_pointcloud` přijímá zprávy `VelodyneScan` a převádí je na obecně používaný formát pro mračno bodů `PointCloud2`,
- `velodyne_laserscan` převádí zprávy tvořené uzlem `velodyne_pointcloud` z jejich 3D podoby do podoby 2D čili do zprávy `LaserScan`, což se hodí v případě, že chceme využívat 2D SLAM algoritmy.

Co se týče parametrů, tak většina z nich jsou samy o sobě vypovídající, proto zde zmíním jen ty, které jsou hodny komentáře:

- `rpm` počet otáček za minutu, musí souhlasit s nastavením senzoru,
- `npackets` senzor produkuje pakety s 3D body a pro shromáždění celé rotace senzoru (záběr 360°), nastavíme parametr na -1 a parametr se vypočte pomocí:

$$npackets = \frac{sensor\_data\_rate}{revolution\_per\_second} \tag{18}$$

[11] `http://docs.pointclouds.org/trunk/classpcl_1_1_v_l_p_grabber.html`

[12] `https://github.com/ros-drivers/velodyne/blob/master/velodyne_pointcloud/launch/VLP16_points.launch`

Pro VLP-16 je datový tok 754 paketů za sekundu a 1508 pro dual return mód. Nutno poznamenat, že v době kdy je tato práce psána, dual return mód není tímto ovladačem podporován. Tento parametr jednoduše říká, kolik paketů se nashromáždí, než se pošle jedna zpráva `PointCloud2`.

- `ring` v uzlu `velodyne_laserscan`, VLP-16 obsahuje 16 dvojic přijímačů a vysílačů laseru. Parametr určuje, ze kterého páru se vezme výstup 3D bodů ze kterých se stane 2D výstup. Pro VLP-16 je volba 0-16, od úplně spodního k nejvyššímu.

Zapnutím *Rvizu*, můžeme sledovat příchozí data. Nastavíme si *fixed frame* podle `frame_id` z uzlu `velodyne_node` (defaultně `velodyne`) a přidáme sledované téma mračna bodů z parametru uzlu `velodyne_pointcloud` (defaultně `velodyne_points`). Pokud příchozí data nejsou vidět, zkontrolujeme jestli máme přidanou směrovací cestu na IP senzoru pro rozhraní, skrze které přicházejí pakety od senzoru. Dalším dobrým testovacím nástrojem, jestli nám data chodí je příkaz `rostopic echo <téma>`.

4.2 Problém s 3D reprezentací

Cartographer sice 3D *SLAM* poskytuje, ale jeho výstupem je projekce do 2D. Pro detekci objektů je potřeba 3D reprezentace světa a k tomu se dá použít dvou způsobů. Pokud se naše zařízení pohybuje jenom na jedné ploše tak se dá využít 2D SLAM pro zjištění aktuální pózy robota. Tyto pózy můžeme použít pro libovolný mapovací 3D framework. Za zmínku stojí:

- *Octomap* viz. sekce 3.4 je nejrozšířenější 3D mapovací framework v *ROSu*, má implementovanou svou vizualizaci, definované speciální zprávy a dokonce ho používá framework *MoveIt*[13] jako svoji 3D reprezentaci světa,
- *PCL* modul octree, který je využit v balíčku *BLAM*,
- *GPU Voxels*[14], voxelová reprezentace na GPU, nicméně pro tento balíček zatím není taková velká *ROS* podpora.

Tento přístup má své problémy. První z nich je, že SLAM probíhá ve 2D, tím pádem naše zařízení se musí pohybovat jen v jedné ploše a nastane problém, jakmile bude plocha nějak zakřivená nebo budeme chtít mapovat třeba více pater v místnosti. Další problém je, když použijeme přímo mapovací framework s pózou ze *SLAMu* a používáme jakýkoliv typ *GraphSLAMu*, tak budeme muset ručně vymazat data z určitých míst podle toho, jak se nám změní uražená trajektorie. Pro tyto důvody jsem se rozhodl nakonec vybrat „úplný“ 3D *SLAM* a implementovat do *Cartographeru* 3D výstup.

[13] `http://moveit.ros.org/`
[14] `http://www.gpu-voxels.org/`

4.3 Seznámení a instalace Cartographeru

Cartographer jsem si nainstaloval pomocí návodu na úvodní stránce dokumentace [30] (na přiloženém CD jsou instrukce upraveny, aby instalace brala v potaz upravenou verzi). Po instalaci *Cartographeru* a jeho závislostí jsem jej mohl vyzkoušet. Vždy, když se s ním pracuje, je třeba mít správně nastavené cesty v prostředí terminálu pomocí souboru `setup.bash`. Jakmile jsem měl *Cartographer* připravený, tak jsem si vyzkoušel data od *Cartographer* týmu dostupná z [30]:

```
roslaunch cartographer_ros demo_backpack_3d.launch bag_filename:=/path/to/bag
```

Cartographer pro 3D *SLAM* potřebuje mít na vstupu mračno bodů, správný model robota, data z *IMU* jednotky a správnou konfiguraci. Pro úvodní živé testování toto představovalo problém, protože jsem neměl žádný dostupný *IMU* čip, který by data produkoval, natož převáděl do *ROS* formátu.

První konfigurace

Pro správné spuštění *Cartographeru* jsou tři hlavní konfigurační soubory:

1. `.launch` – soubor se správným mapováním témat a argumentů pro spuštění uzlů,
2. `.urdf` – soubor, který popisuje model robota, umístění a naklonění senzorů a id rámců sbíraných dat,
3. `.lua` – konfigurační soubor *Cartographeru*.

Inspiroval jsem se soubory přímo v projektu `cartographer_ros`, jež obsahuje konfiguraci pro demo `bag` záznamy.

Launch

Jeho cílem je popsat konfiguraci uzlů. V příloze B výpisu č. 5 je ukázka takové konfigurace. Spouští se pomocí `roslaunch` a vyžaduje argument `bag_filename`, jehož hodnota je cesta k `bag` záznamu. Požaduje existenci souborů `cart_test.urdf`, `cart_test.lua` a `demo_3d.rviz` uvnitř nainstalovaného balíčku `cartographer_ros`. Konfigurace také přejmenovává názvy témat od uzlu `cartographer_node`, jejich defaultní názvy jsou v dostupné v dokumentaci [30]. Launch soubor dále nastaví parametr `/use_sim_time` na `true`, což způsobí, že získávání času z *ROSu* bude vracet čas nulový, dokud nepřijdou data z `/clock` tématu. V tomto případě je `/clock` téma posíláno uzlem `play` z `rosbag`, který spouští přehrávání záznamu `bag` souboru [31].

URDF

Soubor mj. definuje, jaké transformace mezi rámci bude uzel `robot_state_publisher` zveřejňovat. Tato konfigurace, která je v příloze B výpisu č. 6, popisuje transformace mezi rámci `base_link` a `velodyne` a mezi `base_link` a `imu_link`. Dále soubor definuje fyzické parametry robota, spojů a jejich pozic či naklonění. V tomto příkladu jsou fyzické parametry zanedbány až na model VLP-16.

Lua

Tímto souborem se konfiguruje nastavení parametrů *Cartographeru*. Pro první spuštění je hlavní v tomto souboru zkontrolovat:

- rámce `tracking_frame` a `published_frame`,
- hodnotu parametru `num_point_clouds`, která musí být jedna, protože máme pouze jeden zdroj mračna bodů,
- parametr `TRAJECTORY_BUILDER_3D.num_accumulated_range_data`

Ten souvisí se vzorcem (18). Pokud použijeme nastavení *Cartographer* týmu a necháme `velodyne_driver` produkovat jednu zprávu za jeden paket, tak hodnotu parametru nastavíme na 80 (v nastavení od Google je 160 pro dva VLP-16, pro jeden odpovídá zhruba hodnotě 75.6). Jestliže chceme omezit množství zpráv, můžeme nastavit `npackets` na větší hodnotu, ale potom podle toho správně nastavit parametr `num_accumulated_range_data`, aby odpovídal zhruba hodnotě za celý 360° sken. Pokud chceme posílat jednu zprávu pro jeden 360° sken, tak nastavíme `npackets` na -1 a parametr `num_accumulated_range_data` na 1.

4.3.1 Experimenty s IMU

Pro první živý test, jsem neměl dostupný samostatný čip *IMU*. Protože dnešní chytré telefony obsahují jak akcelerometr tak gyroskop, vyhledal jsem aplikaci na Android, která tyto data sbírá a přeposílá ve formátu *ROS* přihlášeným odběratelům.

V sekci 3.2 bylo zmíněno že, *ROS* podporuje více jazyků a jeden z nich je Java společně v kombinace s Androidem. Projekt *ROS Driver for Android Sensors* [32], využívá *ROS* API a *Android* API pro čtení senzorů z telefonu. Projekt jsem si tedy naklonoval, nainstaloval potřebné *Android* SDK a poté zkompiloval a vytvořil `apk` soubor. Aplikace (očekávaně) občas potěší pádem, ale pro většinu času funguje korektně.

Rviz defaultně nabízí vizualizaci *IMU* zprávy (sekce 3.2.5), ale lepší vizualizaci nabízí balíček *imu_tools*[15], tato vizualizace se skvěle hodí pro testování různých senzorů.

[15] http://wiki.ros.org/imu_tools

Synchronizace androidích zpráv IMU

Pro usnadnění práce jsem si před sestavením aplikace přejmenoval id rámce na `imu_link` (změna ve třídě `ImuPublisher` pro volání metody `setFrameId`). Při prvním sběru dat s telefonem jako *IMU* se začalo objevovat varování ve výpisu č. 1 a žádná data nebyly v *Rvizu* vidět.

```
...
[ WARN] [1512074277.194455759, 1511821311.204946129]: W1130 21:37:57.000000
    22061 ordered_multi_queue.cc:155] Queue waiting for data: (0, points2)
[ WARN] [1512074277.227861131, 1511821311.236593011]: W1130 21:37:57.000000
    22061 ordered_multi_queue.cc:155] Queue waiting for data: (0, points2)
...
```

Výpis 1: Výpis Cartographeru při problému s telefonem IMU

I přesto, že *SLAM* nefungoval, jsem si nahrál data do formátu `bag` pomocí příkazu:

```
rosbag record /velodyne_points /android/tango1/imu
```

Zároveň jsem zachytával *UDP* pakety od VLP-16, pomocí programu *Wireshark* do `pcap` souboru, abych později mohl simulovat synchronizaci s *IMU*, i když nebudu mít u sebe dostupný senzor.

Díky zaznamenaným souborům `bag` se nakonec se ukázalo, že chyba byla v časovém údaji v hlavičce zpráv z telefonu. Celý problém je poté popsán i diskusí na Githubu *Cartographeru*[16]. Jedno řešení je buď napsání vlastního časové filtru, který vždy příchozí *IMU* zprávě z Androidu přiřadí čas stroje, ale poté by tento časový údaj nebyl korektní. Nakonec jsem modifikoval zdrojový kód androidí aplikace. Provedené změny jsou poté vidět ve výpisu níže.

```
/*  Original
*  long time_delta_millis = System.currentTimeMillis() - SystemClock.
    uptimeMillis();
*  this.imu.getHeader().setStamp(Time.fromMillis(time_delta_millis + event.
    timestamp / 1000000));
*   this.imu.getHeader().setFrameId("/android/imu");// TODO Make parameter
*/
   //edited
    this.imu.getHeader().setStamp(Time.fromMillis(System.currentTimeMillis());
    this.imu.getHeader().setFrameId("/imu_link");
```

Výpis 2: Úprava zdrojového kódu třídy ImuPublisher v aplikaci ROS Sensors drivers

Po těchto úpravách SLAM začal fungovat s občasným hlášením výše uvedeného varování, protože rychlost odesílání telefonního *IMU* není pro *Cartographer* optimální. Pro první testování, ale je tato metoda dostačující.

[16] https://github.com/googlecartographer/cartographer/issues/722

Nasazení IMU čipu

Tým *Cartographeru*, pro své účely používali *IMU* Xsens MT10 a MT-1[17]. Po problémech s telefonem jsem dokoupil čip InvenSense MPU9255[18]. Tento čip jsem přes sběrnici I^2C připojil k Raspberry Pi 3 (dále jen *RPI*).

Na *RPI* bylo třeba nainstalovat *ROS* s ovladačem, který bude zpracovávat data přicházející z MPU9255 po I^2C. Pro zjednodušení práce jsem si zvolil obraz systému, který již *ROS* ve verzi Kinetic má nainstalovaný[19]. Pro *IMU* čip jsem použil ovladač pro *ROS* `i2c_imu` dostupný z GitHubu[20]. Tento ovladač je jen most mezi *ROSem* a mezi *RTIMULib2*[21], která zpracovává data z čipu. Nainstaloval jsem tedy podle instrukcí[22] *RTIMULib2*, kde musím poznamenat, že jsem instrukce pro *RPI* vynechal a jen jsem knihovnu sestavil a nainstaloval. Testovací program `RTIMULibDemo` fungoval korektně pod `sudo`, což bylo dostačující.

Při testování se objevil problém. Přes marnou snahu vyladit parametry *Cartographeru*, se zdálo, že výsledná mapa má tendenci se otáčet na druhou stranu. Proto jsem si nahrál data ze dvou *IMU* výstupů (jeden z androidí aplikace a druhý z čipu) a poté jsem provedl analýzu pomocí `rqt_bag`. Výsledkem bylo, že osa *Z* měla podobné hodnoty úhlových rychlostí, ale na opačnou stranu. Následně jsem testoval různé typy souřadnicových systémů, které jdou nastavit v *RTIMULib2*, ale ani jeden neodpovídal. Nakonec jsem upravil kód ovladače `i2c_imu`, kde jsem výsledek úhlové rychlosti pro osu z vynásobil -1.

Je třeba dodat, že senzory *IMU* produkují pouze lineární zrychlení a úhlové rychlosti. Zpráva `sensor_msgs/Imu`[23], obsahuje i orientaci. Ta se obvykle získává tzv. fusion algoritmem, který spojí výsledky z senzorů co *IMU* jednotka má. Knihovna `i2c_imu` i androidí aplikace tyto algoritmy poskytují. Pro analýzu dalších čipů jsem se rozhodl využít uzel `imu_complementary_filter` v balíčku `imu_tools`, aby výsledná orientace ve zprávě byla tvořena stejným algoritmem.

Konfigurace vzdáleného spuštění

ROS umožňuje jedním `launch` souborem spustit uzly na více počítačích. Na lokálním stroji pustí uzly klasicky a na vzdáleném stroji je pustí pomocí `ssh`. Aby se *ROS* mohl na vzdáleném zařízení autorizovat (v mém případě *RPI*), musí se správně nastavit `ssh` klíče. Díky tomu, že *ROS* používá knihovnu, která nepodporuje aktuální volbu klíče (viz.[24]), tak ve výpisu č. 3 jsou kroky pro správnou autorizaci *ROSu*.

[17] `https://groups.google.com/forum/#!topic/google-cartographer/sMLkPVAnaB4`
[18] `https://stanford.edu/class/ee267/misc/MPU-9255-Datasheet.pdf`
[19] `http://www.german-robot.com/2016/05/26/raspberry-pi-sd-card-image/`
[20] `https://github.com/jeskesen/i2c_imu`
[21] `https://github.com/Nick-Currawong/RTIMULib2`
[22] `https://github.com/RTIMULib/RTIMULib2/tree/master/Linux`
[23] `http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html`
[24] `https://github.com/ros/ros_comm/issues/612`

```
IP_RPI = 192.168.0.101
#vygenerování privátního a~veřejného klíče
ssh-keygen
ssh-copy-id $IP_RIP
#následně musíme odstranit otisk RPI z ~/.ssh/known_hosts
#což bude poslední záznam, jinak bude pokus o~připojení hlásit chybu
ssh -oHostKeyAlgorithms='ssh-rsa' pi@$IP_RPI
#další připojení již nemusí specifikovat algoritmus klíče
```

Výpis 3: Příkazy potřebné pro autorizaci ROS pro vzdálené spuštění

Dále je třeba skript, který připraví proměnné prostředí, jako je třeba URI mastera, IP stroje, kde bude uzel běžet a cesty k balíčkům. Tento skript jsem napsal tak, aby se IP mastera nastavilo podle toho, kdo se připojuje přes `ssh`. Pokud chceme použít toto vzdálené spuštění, musí být i na lokálním počítači nastavená proměnná prostředí `ROS_IP`. Skript je obsahem CD.

V mém případě jsem napsal `launch` soubor, aby na *RPI* spustil uzel pro odesílání *IMU* zpráv. Také je třeba dbát na to, aby se *RPI* měl synchronizovaný čas s počítačem, který produkuje zprávy mračna bodů. Pokud počítače nemají přístup na internet, dá se využít *NTP Orphan mode*[25], což jsem nastavil jak na *RPI* tak na můj používaný počítač.

Ladění lokálního SLAMu

Mapa po opravě rotační rychlosti v ose z pro zprávy *IMU* už vypadala lépe, ale i tak bylo třeba další konfigurace. V sekci 3.3.2 popisuji, že *Cartographer* má lokální *SLAM* a globální *SLAM*. Pro správnou funkci globálního *SLAMu* musí být dobře nakonfigurovaný *SLAM* lokální. Dokumentace doporučuje pro ladění lokálního *SLAMu* vypnout ten globální. Ten se vypíná parametrem

```
POSE_GRAPH.optimize_every_n_nodes = 0
```

Tímto parametrem se také určuje, jak často proběhne optimalizace grafu na pozadí. Většina parametrů pro konfiguraci lokálního SLAMU je v `trajectory_builder_3d.lua`.

Hlavní parametry pro správnou funkčnost jsou `translation_weight` a `rotation_weight`. Když se hledá optimální póza pro vložení nových dat do submapy, tak se scan-matcher v ní snaží maximalizovat pravděpodobnosti. Čím vyšší tyto váhy jsou, tím se musí najít lepší shoda mezi nově vloženým skenem a mezi záznamem v submapě, aby nová póza byla přijata. Pro můj případ jsem experimentem nalezl hodnoty pro tyto váhy. Nalezené hodnoty byly `translation_weight` $= 8$ a `rotation_weight` $= 20$. Další podstatný parametr je velikost submap. Dle dokumentace je třeba mít správně nastavenou velikost submapy, aby chyba v ní nebyla větší než je nastavené rozlišení (parametr `high_resolution` a `low_resolution` v metrech) a aby

[25] `http://support.ntp.org/bin/view/Support/OrphanMode`

byla submapa dostatečně unikátní pro korektní funkčnost loop closure. Vyhodnotil jsem, že je v pořádku a nechal jsem velikost submapy na původní hodnotě. Toto se dá hezky vizuálně ověřit v *Rvizu*, kde je seznam submap a dá se vybrat, které chceme vizualizovat (ve výchozím nastavení se vizualizují všechny).

Ladění globálního SLAMu

Při nevyladěném lokálním *SLAMu* a vypnutém globálním, vypadala mapa v pořádku až na pár nepřesností. Když byl zapnutý špatně vyladěný globální *SLAM*, tak „optimalizovaná“ trajektorie byla kompletně mimo. Zapnutí globálního *SLAMu* ukázalo, že stávající popis *URDF* není dostačující. Upravil jsem tedy model auta, aby více odpovídal realitě. Problém byl, že začáteční póza je v $(0, 0, 0)$ a *Cartographer* umisťoval pózy grafu ve výši senzoru VLP-16, což poté mylně vytvářelo trajektorii „do kopce“, i když auto jelo rovně. Proto jsem konfiguraci *URDF* přepsal tak, aby samotný senzor začínal v pozici $(0, 0, 0)$ a pohybem auta zůstal ve stejné rovině. Toto způsobuje, že podlaha nebude na nulté hodnotě osy z. Samotný model auta jsem zjednodušeně popsal jako krabici o jeho reálných rozměrech a zmenšil její výšku o velikost kol.

Při ladění parametrů jsem bral v úvahu, že hlavní problém v lokálním *SLAMu* byl `rotation_weight`, nejspíše plynoucí z *IMU* horší kvality. Proto jsem se zaměřil na váhové parametry. Skutečně se ukázalo, že hlavní problém byl parametr `rotation_weight`, který určuje váhu rotace z *IMU* dat. Nakonec jsem tento parametr vyladil na hodotu 30. Další parametry, které jsem ladil a stojí za zmínku:

- `optimize_every_n_nodes` – říká po kolika vložených skenech se spustí globální *SLAM*. Nemá smysl pouštět více než 1x za submapu, proto ho nastavuji na počet skenů v submapě,
- `constraint_builder.min_score` – když najde scan–matching sken, který sedí s jiným, tak mu *Cartographer* přiřadí skóre. To je v procentech a určuje, jak moc si skeny jsou podobné. V podobných prostředích, jako je podlouhlá chodba se mi osvědčilo nastavit tento parametr na vyšší hodnotu (0.6 osvědčené na chodby v budově škole),

5 Implementace

Pro detekci překážek je potřeba 3D prostorové reprezentace. V minulé sekci jsme se dozvěděli, že *Cartographer* a jeho 3D *SLAM* nemá výstup pro 3D data. Zde se dozvíme, co jsem musel udělat pro takovou implementaci. *Cartographer* má mapu rozdělenou na menší části a proto nejde využít hotový framework pro plánování cest a kontrolu překážek, který počítá s celistvou mapou. Mapa by se musela spojit dohromady a to je značně neefektivní, nejenom paměťově, ale i výpočetně. Vzhledem k tomu, že se pózy submap budou měnit, musely by se části mapy navíc mazat. Je nutno poznamenat, že pro implementaci jsem vycházel z verze *Cartographeru* 0.3.0. Jakmile jsem měl 3D výstup, implementoval jsem vlastní balíček `obstacle_detection`, který kontroluje místo před nebo za autem, podle toho, jakým směrem auto jede, pomocí vysílání paprsků do prostorové reprezentace mapy.

5.1 Získávání 3D dat z Cartographeru

3D *SLAM Cartographeru* má na výstupu jen projekci 3D bodů do roviny a proto jsem musel tuto implementaci napsat sám[26]. Musel jsem se rozhodnout jestli získávání 3D dat napíšu zvlášť do svého projektu nebo rozšířím `cartographer_ros`. Rozhodl jsem se pro druhou variantu, protože jsem zaregistroval, že se po této funkci ptalo více lidí na diskuzích o *Cartographeru* a protože balíček `cartographer_ros` již nabízel všechny dostupné prostředky k této funkcionalitě. Následoval jsem styl pro získávání 2D dat a vytvořil jsem službu nazvanou `SubmapCloudQuery`, která bude vracet 3D mapu jedné submapy, jejíž definice je ve výpisu č. 4. První polovina určuje parametry služby:

- id trajektorie a index submapy určují unikátní identifikátor submapy,
- `min_probability` umožní zmenšit množství přenesených dat a parametr,
- `high_resolution` určí jestli získáváme data ve vysokém či nízkém rozlišení.

```
int32 trajectory_id
int32 submap_index
float32 min_probability
bool high_resolution
---
int32 submap_version
float32 resolution
bool finished
sensor_msgs/PointCloud2 cloud
```

Výpis 4: Definice služby SubmapCloudQuery

[26] `https://github.com/googlecartographer/cartographer_ros/issues/720`

Druhá polovina výpisu říká návratové hodnoty. Jmenovitě verzi mapy, která určuje kolik skenů je do ní vloženo a `finished` říká, jestli se ještě do submapy budou přidávat další skeny. V parametru `cloud` jsou přenášeny samotná data v podobě centroidů voxelové mřížky. Tuto reprezentaci jsem zvolil, protože není závislá na konkrétním frameworku a klient si ji může převést do libovolné 3D reprezentace.

Převod dat do ROS zprávy PointCloud2

Cartographer 3D data poskytuje jako voxely, které mají velikost v závislosti na jejich rozlišení. Balíček `cartographer` 3D data poskytuje skrze volání metody `SubmapData::ToProto` na konkrétní submapě. Ta se dá získat ze třídy `PoseGraph` a tu obsahuje třída `MapBuilder`. V projektu `cartographer_ros` k ní má přístup `MapBuilderBridge`, proto je hlavní získávání dat implementováno zde metodou `HandleSubmapCloudQuery`.

Získaná data ze submapy obsahují indexy voxelů v prostoru a pravděpodobnostní hodnotu jejich obsazenosti. Výstupem `SubmapCloudQuery` jsou centroidy těchto voxelů. Převod mezi voxely a centroidem voxelu je:

$$centroid\_position = indices\_vector \cdot submap\_resolution + \frac{submap\_resolution}{2} \tag{19}$$

Každé pole voxelové mřížky zároveň prochází filtrací na její pravděpodobnostní hodnotu, přičemž maximální hodnota pravděpodobnosti je 2^{15}. Zároveň do pole `intensity` každého bodu ukládám jeho pravděpodobnost, kdyby s ní klient chtěl dále pracovat nebo ji vizualizovat.

5.2 Implementace balíčku obstacle_detection

Jakmile jsem měl 3D výstup hotový, tak jsem vytvořil vlastní balíček `obstacle_detection`. Použil jsem příkaz `catkin_create_pkg`, který vytvoří základní adresářovou strukturu, aby se balíček hned dal sestavit a použít. Poté jsem upravil `package.xml` a `CMakeLists.txt`, aby se reflektovaly závislosti z kódu, aby se správně vygeneroval `Makefile` a aby při instalaci byly soubory umístěny do požadovaných složek. Do balíčku jsem přesunul všechny konfigurační soubory a pro zjednodušení jsem je dal do hromadné `config` složky. Do složky `launch` jsem přesunul stejnojmenné soubory a upravil v nich cesty. Pro implementaci jsem si zvolil jazyk C++, protože se jedná o hlavní programovací jazyk pro *ROS* a pro jeho integraci s knihovnami jako je *PCL* nebo *Octomap*.

Balíček jsem tvořil v tzv. catkin workspace, což je složka, která sdružuje balíčky a umožňuje jejich snadné sestavení a instalaci. Vytvoření workspace je ve skutečnosti pouhé vytvoření složky s jakýmkoliv názvem, vytvořením podsložky `src` a spuštěním příkazu `catkin_make` ve složce workspace. Jakmile jsou balíčky v podsložce `src`, tak stejný příkaz balíčky projde, spustí nad nimi `cmake`, jenž vygeneruje `Makefile` soubory v podsložce `build` a poté spustí samotný `make`. Výsledné knihovny a spustitelné soubory poté budou v podsložce `devel` a při instalaci ve složce `install`.

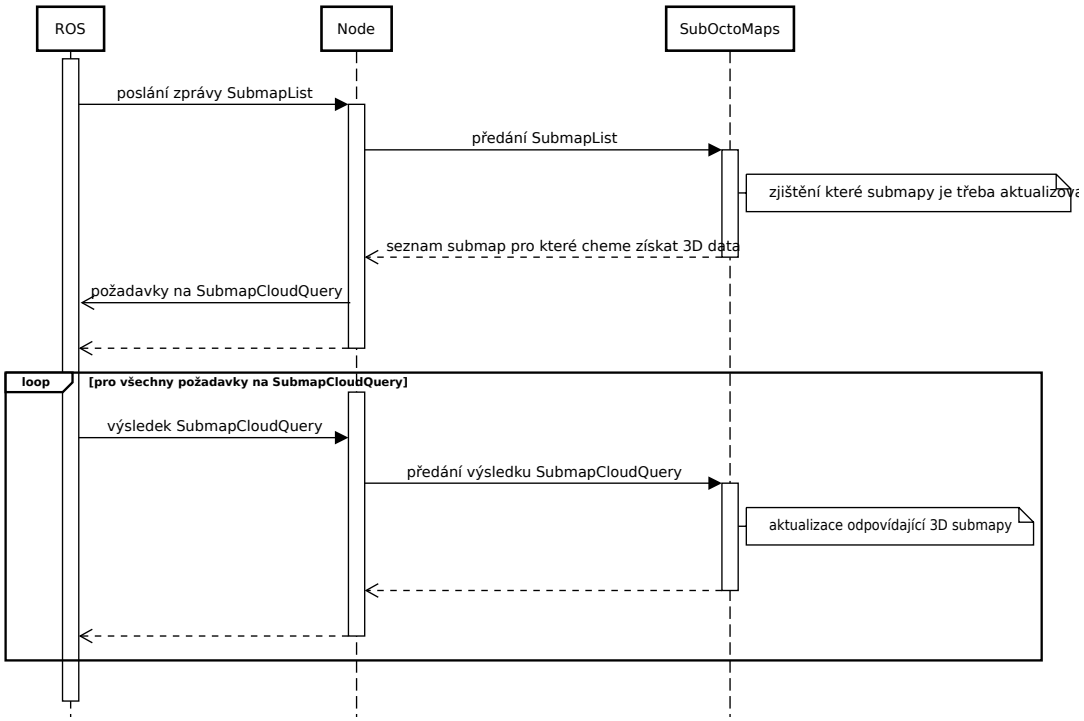
Při používání balíčků jsem měl dva workspace, jeden pro *Cartographer* a jeden pro vlastní balíček. Je třeba dát pozor, aby před spuštěním příkazu `catkin_make` byly v terminálu cesty ze skriptu `install/setup.bash` z workspace s instalovaným *Cartographerem*. Tímto způsobíme, že druhý workspace již počítá s cestami z prvního workscape a díky toho, nám stačí při používání mít nastavené cesty z jednoho skriptu `install/setup.bash`. Tomuto principu se říká *workscape overlaying* [33].

5.2.1 Integrace ROS a Cartographeru

Klíčové pro tuto implementaci bylo vytvořit kód, který bude správně komunikovat s *ROSem*. Pro vytvoření vlastního uzlu, jej stačí zaregistrovat jako spustitelný soubor do `CMakeLists.txt` a ve funkci `main` uzel inicializovat voláním metody `ros::init`. Dále pomocí `ros::NodeHandle` můžeme získat parametry, jejichž konfiguraci můžeme změnit buď v `launch` souboru nebo přímo při spuštění.

Postup zpracování dat

Jakým způsobem se se data zpracovávají je vidět na obrázku č. 12. *Cartographer* publikuje zprávu typu `SubmapList`, která obsahuje seznam všech submap a jejich póz. Třída `Node` má zaregistrovaný callback, na příchozí zprávu tohoto typu a každou z nich předá třídě `SubOctoMap`, která má za cíl si udržovat 3D reprezentaci submap pro kontrolu kolizí. Ta podle konkrétní implementace vybere, které potřebuje aktualizovat. Dodaná implementace si udržuje aktuální posledních N submap, kód je ale navržen tak, aby fungoval pro libovolný princip udržování submap.



Obrázek 12: Sekvenční diagram, jenž naznačuje integraci dat ve vlastním balíčku, odesílané *Cartographerem* skrze *ROS*

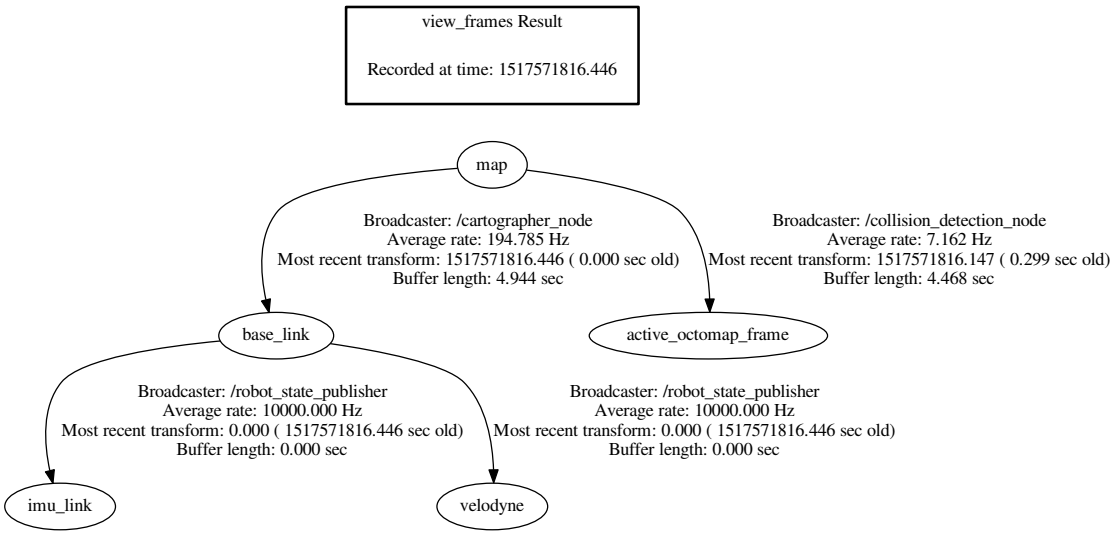
Protože 3D data nejdou bez použití služby vidět, tak jsem vytvořil jednoduchý Python uzel, který získává a publikuje submapu dle jejího identifikátoru a dle parametru `min_probability`. Uzel z konzole se dá pustit pomocí příkazu

```
rosrun obstacle_detection cartographer_submap_tester _submap_index:=1
```

Na výše uvedeném příkazu můžeme vidět předávání parametrů v *ROS* stylu, kde úvodní podtržítko značí, že parametr je privátní, ten je v uzlu získáván s tilda prefixem. Výstup jde sledovat v *Rvizu*, kde jako *fixed frame* je `map`.

Zjištění aktuální polohy robota

Póza robota je pravidelně publikována jako transformace mezi rámcem robota (`base_link`) a rámcem mapy (`map`). V *ROSu* jsou transformace odděleny od obyčejných zpráv a tvoří strom transformací mezi rámci. Takový strom je vidět na obrázku č. 13. Pokud existuje ve stromu cesta z jednoho rámce do rámce druhého, tak poté můžeme získat mezi nimi přímou transformaci, která je tvořena translací a rotací. Transformace jsou také vždy značeny časem a dokonce můžeme získat transformaci i mezi publikovanými časy skrze automatickou interpolaci.



Obrázek 13: Strom transformací při spuštěném *Cartographeru* a mnou implementovaným uzlem

Kontrola kolize

Jak jsem již zmínil v předchozí sekci, 3D data zpracovává třída `SubOctoMaps`, která si uchovává aktuální 3D reprezentaci submap a umožňuje pomocí metody `rayCollision` vyslat paprsek do všech aktivních submap. Takto je kontrola oddělená od konkrétní implementace.

Mapa se skládá z pozic voxelů a jejich pravděpodobnosti. Výstup pro ostatní balíčky jsou centroidy voxelů ve formě mračna bodů, získané ze služby `SubmapCloudQuery`. Není možnost, jak kontrolovat obsazený prostor v obyčejném mračnu bodů. Musí se použít nějaká prostorová reprezentace bodů, jako je třeba oktantový strom reprezentující hierarchickou strukturu prostoru. Na to se nabízí buď modul `octree` z knihovny *PCL* nebo knihovna *Octomap*. Pro moji implementaci jsem zkoušel obě a nakonec jsem ponechal *PCL*, protože její výkon byl viditelně vyšší.

Kolizi zjišťuji vysíláním paprsků ve směru jízdy auta. Protože prostorová reprezentace má konkrétní rozlišení, tak stačí vyslat jen konečný počet vyslaných paprsků. Další možností vyhledávání kolize je projekce robota do jiné polohy na mapě a kontrola počtu bodů v této projekci. Pro sofistikovanější kontroly kolizí se nabízí knihovna *Flexible Collision Library*[27], ale v tomto případě nebyla potřeba.

Kontrola začne probíhat po spuštění v samostatném vlákně až do ukončení uzlu. Jak často se kontrola provádí závisí na parametru `detection_rate_hz`, který říká, kolikrát za sekundu by měla být kontrola kolizí spuštěna. Abych nemusel v programu nastavovat parametry auta

[27] `http://gamma.cs.unc.edu/FCL/fcl_docs/webpage/generated/index.html`

explicitně a udržovat je na dvou místech tak je získávám z načteného `urdf` souboru, který je dostupný v globálním parametru `/robot_description`.

Podle velikosti auta a rozlišení mapy vytvořím takový počet paprsků, který pokrývá přední nebo zadní část auta. Počet paprsků tedy odpovídá

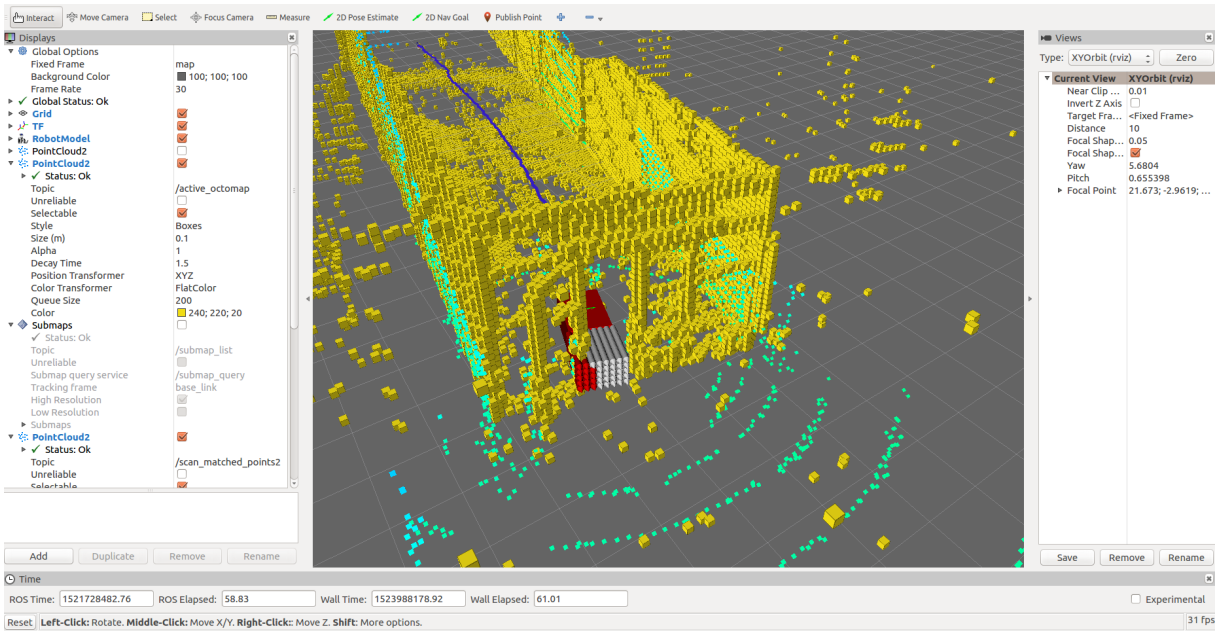
$$rays\_count = \frac{car\_width \cdot car\_height}{map\_resolution^2}. \tag{20}$$

Paprsky si před začátkem vygeneruji zvlášť ve statickém rámci, kde je model auta umístěný na pozici $[0, 0, 0]$, díky toho paprsky směřují vždy před nebo za auto.

Pro samotnou kontrolu souřadnice paprsku je převedu do globálního rámce a zároveň je brán v potaz směr auta, který zjišťuji rozdílem mezi předchozí a aktuální pózou. Jakmile se paprsek dostane do metody submapy, tak ho převedu pomocí pózy submapy do statického rámce. Díky toho, že každá submapa má body právě ve statickém rámci, tak další transformace nejsou potřebné. Pak už stačí provést samotnou kontrolu, pro ni využívám modul *octree* z *PCL*.

5.3 Vizualizace

Pro kontroly kolizí má každý paprsek vlastní vizualizační prvek, v tomto případě šipku. Dále také aktuální submapy publikuji jako mračno bodů, do zvláštního rámce `active_octomap_frame` a společně s nimi publikuji transformaci do rámce mapy. Šipky jsou normálně šedé a jakmile hrozí kolize, tak zčervenají a zároveň se kolize vypíše do konzole. Vizualizace je na obrázku č. 14.



Obrázek 14: Vizualizace kolize a 3D mapy v programu *Rviz*

6 Testování

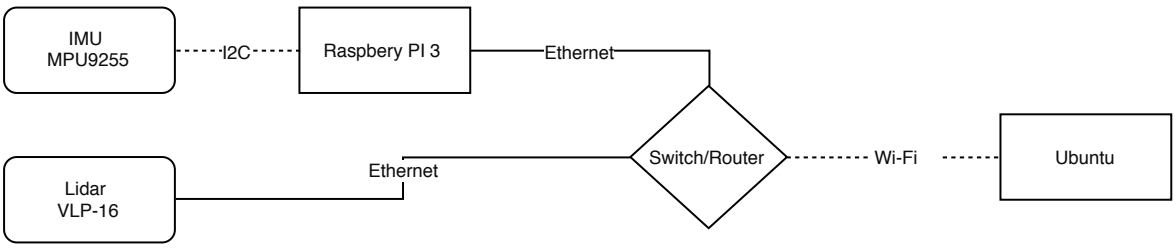
Testy probíhaly zároveň s implementací a konfigurací. Pro první testovací prostředí poskytl své služby obyčejný vozík s krabicí, na kterém byl umístěn VLP-16. Na něm byly zjištěny problémy s *IMU*, které jsem zmiňoval v sekci 4.3. Poté, co chyba způsobená čipem *IMU* byla opravena, jsme postupně sestavili auto, které se stalo posledním testovacím prostředím. To můžeme vidět na obrázku č. 17. V této sekci se dozvíte, jaké testy byly s tímto autem provedeny s ukázkou vytvořené mapy při pohledu shora.

6.1 Popis testovacího prostředí

Pro sumarizaci celé testovací prostředí obsahovalo:

- lidar VLP-16, umístěný na střeše auta a připojený skrze Ethernet k switchi/routeru,
- *IMU* čip MPU9255, který je skrze I^2C, připojen na *RPI* 3,
- *RPI* 3, který sloužil jen jako most mezi *IMU* a *ROSem* běžící na Ubuntu,
- počítač s operačním systémem Ubuntu 16.04 LTS, ve kterém je nainstalovaný *ROS*. V mém případě to byl virtuální stroj s 8 GiB paměti na notebooku s procesorem Intel Core i7-7700HQ.

Blokovém schématu na obrázku č. 15 značí, jak byly zařízení mezi sebou propojeny.



Obrázek 15: Blokové schéma testovací prostředí

Auto bylo ovládáno vzdáleně pomocí aplikace v telefonu skrze *Bluetooth*. Jedno z možných vylepšení, je nenechat router propouštět globální broadcasty z VLP-16 skrze Wi-Fi, ale jen je přeposlat do *RPI*. Na něm může běžet uzel ovladače VLP-16, který bude publikovat mračno bodů jako *ROS* zprávu. Toto zapojení by mělo zajistit lepší kvalitu 360° skenu, protože bude v režii *ROSu*. Momentálně totiž skeny občas „vynechávají“, což může způsobit horší scan-matching. Nicméně i přes tento problém ve všech naměřených datech při posledním nastavení byl loop closure nalezen.

Další vylepšení by mohlo být, přepsání aplikace, která auto ovládá tak, aby generovala odometrii, jakožto další vstup pro *Cartographer*.

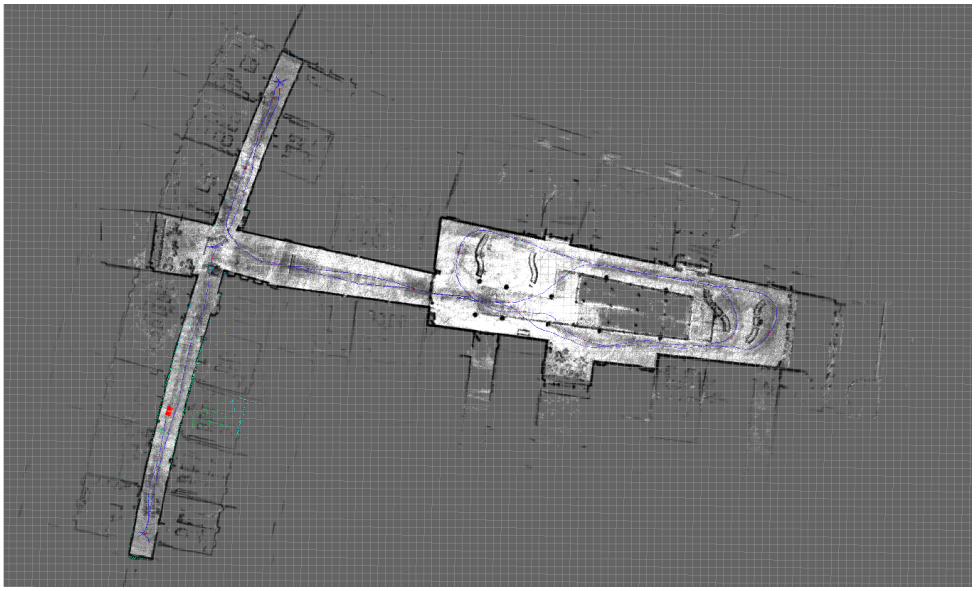
6.2 Testovací sady

Protože byl využit *ROS*, bylo snadné udělat záznamy z testovacích jízd. K tomu stačil příkaz `rosbag record <téma1> <téma2>`. Každá testovací sada reprezentuje jednu jízdu se zmíněným autem po budově Fakulty elektrotechniky a informatiky VŠB-TUO (dále jen *FEI*). Testovací sady jsou:

- `dod.bag` je záznam ze dne otevřených dveří FEI, na kterém jsme zaznamenávali druhé patro budovy FEI, ve kterém bylo relativně velké množství lidí, které teoreticky kazí scan-matching, ale i přes to *Cartographer* byl schopen mapu po určité době spravit.
- `cele_patro.broken.bag` tento záznam je označen jako broken, protože byl pořízen, když virtuální stroj měl málo místa na disku a proto se záznam neuložil korektně. I přesto, poté co jsem na něm spustil příkaz `rosbag reindex`, tak začal fungovat. Jedná se o záznam celého druhého patra VŠB FEI v odpoledních hodinách.
- `druhe_patro_1.bag` záznam druhého patra u počítačových laboratoří,
- `druhe_patro_2.bag` opět záznam druhého patra, tentokrát jsme se s autem vydali do prostoru před přednáškovými místnostmi. Ke konci záznamu *IMU* začal produkovat špatné hodnoty, ale po kontrole připojení *IMU* a znovu spuštění, začalo být všechno v pořádku. Proto se v tomto záznamu nedojedeme až na start cesty a poslední submapa nesedí, nestihne se totiž spustit včas optimalizace.

Pro ukázku mapování většího místa, můžeme sledovat výsledek naší jízdy z `cele_patro.broken.bag` na obrázku č. 16. Ostatní mapy jsou v příloze D. Každý záznam se spouští příkazem

```
roslaunch obstacle_detection cart_3d_bag.launch bag_filename:=/path/to/bag
```



Obrázek 16: Výsledek mapování druhého patra na budově FEI

Pokud bychom chtěli provést živý test, tak použijeme

```
roslaunch obstacle_detection cart_3d_remote.launch
```

což nám spustí uzel pro *IMU* na *RPI*, *SLAM* na počítači a uzel na zpracovávání lidarových dat v jednom příkazu.

Testování bylo také vyzkoušeno na samotném *RPI*, ale spíše jen pro zjištění, jestli vůbec tato konfigurace bude fungovat. Všechen potřebný software na něm fungoval a mapa byla opravována, jen se celý systém zdál zpomalený, což je očekávané vzhledem k výkonu *RPI*. Nicméně si myslím, že *Cartographer* nabízí tolik nastavení, že pokud bychom chtěli celou konfiguraci rozjet pouze na *RPI*, tak s určitým zhoršením kvality (např. zvětšení rozlišení map, méně iterací a jader pro optimalizaci grafu, zvětšením filtrace mračna bodů) by rychlost počítače *RPI* byla dostačující.

Na závěr je nutno poznamenat, že parametry jsem vyladil tak, aby vyhovovaly všem pořízeným datům a aby se v nich správně nalezly loop closures. Vzhledem k velikosti záznamů (nejmenší má `dod.bag`, který má 2.8 GiB) tyto soubory jsem předal vedoucímu práce a nejsou součástí přílohy.

Testování na veřejně dostupných datech

Protože *Cartographer* vyžaduje pro 3D *SLAM* synchronizaci *IMU* a mračna bodů, bylo testování provedeno na jejich datech [30]. Nicméně v jejich případě byl robot batoh nošený na zádech. Proto tyto testy nejsou moc směrodatné. Pro otestování správné funkčnosti kolizí jsem tento test spouštěl s větší hodnotou parametru `ray_max_range`. Test musí mít jejich konfigurace, které jsem zahrnul do balíčku `obstacle_detection` a spouští se pomocí `demo_backpack_3d.launch`.

7 Závěr

Dnešní zájem veřejnosti ohledně tématu samořiditelných aut se zvedá a proto je zajímavé zjišťovat, jak taková auta fungují a jaké technologie využívají. V této práci jsem se zabýval problematikou detekce překážek, která je jedna ze základních funkcí samořiditelného auta. Reprezentace prostředí ve kterém detekce překážek probíhá, si robot tvoří za jízdy, pomocí různých senzorů a jeden z populárních je lidar, který využívá laserového paprsku k zjištění vzdálenosti od povrchu. Takový senzor byl použit i v této práci.

Bylo řečeno, že takovou tvorbou map se zabývá problém současné lokalizace a mapování *SLAM* a první vědecké články tento problém popisují již od konce 90. let minulého století. V práci byl problém detailněji rozepsán a také zde byly popsány tři rodiny řešení tohoto problému společně s náznakem jejich implementace.

V praktické části byl použit software *Google Cartographer*, jakožto jedna z novějších implementací *SLAM* řešení. Používá i nejnovější přístup řešení *SLAM* problému na základě optimalizace grafu. *Cartographer* byl používán společně s *ROSem*, o kterém jsme si řekli, k čemu je dobrý a že jedna z hlavních výhod *ROSu* je oddělení komponent systému od sebe.

Během práce bylo vytvořeno testovací prostředí. Jednalo se o zmenšený model auta, které mělo na sobě namontovaný lidarový senzor VLP-16, mini-počítač *Raspberry PI 3*, který zpracovával data z *IMU* jednotky MPU9255 a počítačem s *ROSem*, ve kterém probíhal *SLAM*. *Cartographer* byl pro toto prostředí patřičně nakonfigurován. Následně v práci uvádím, jaké parametry byly třeba vyladit, aby tvoření mapy společně s loop closure fungovalo korektně. Prostředí také bylo navrženo tak, že se dá celé spustit jedním příkazem.

Cartographer dělí prostor na menší části, které mají místo na mapě. Zároveň její místo na mapě průběžně mění a proto nešlo použít hotová řešení pro detekci překážek a plánování pohybu. Dokonce *Cartographeru* chyběl výstup pro 3D reprezentaci světa. Podle oficiálního diskuzního fóra *Cartographeru* se po této vlastnosti ptalo více lidí, proto tato funkce byla v práci implementována přímo do *Cartographeru*. Následně tento 3D výstup je využit v vlastním balíčku, který si udržuje část tvořené reprezentace mapy. V tomto vlastním balíčku je implementován detektor překážek za jízdy pomocí vysílání paprsků do prostoru. Další část implementace by mohla zahrnovat plánovač cest a lepší model auta v *ROSu* společně s modelováním pohybu kol.

Navržené řešení bylo během implementace testováno ve vytvořeném testovacím prostředím. Během práce byly pořízeny záznamy budovy nové Fakulty Elektrotechniky a Informatiky VŠB-TUO. Na těchto datech bylo prováděno testování navržené implementace. Test proběhl i na veřejně dostupných datech. Testovací prostředí není konečné v sekci testování jsem navrhl dvě možná vylepšení, které by mohly ovlivnit kvalitu tvořené mapy a zároveň by umožnily ovládání auta skrze *ROS*, což by umožnilo plánovači cest i ovládání.

8 Literatura

[1] SMITH, Randall; SELF, Matthew; CHEESEMAN, Peter. A stochastic map for uncertain spatial relationships. 1987, s. 467–474. ISBN 0-262-02272-9.

[2] MOUTARLIER, Philippe; CHATILA, Raja. An experimental system for incremental environment modelling by an autonomous mobile robot. In: HAYWARD, Vincent; KHATIB, Oussama (ed.). *Experimental Robotics I*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, s. 327–346. ISBN 978-3-540-46917-9.

[3] HESS, Wolfgang; KOHLER, Damon; RAPP, Holger; ANDOR, Daniel. Real-Time Loop Closure in 2D LIDAR SLAM. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, s. 1271–1278.

[4] *Lidar UK* [online] [cit. 2018-03-31]. Dostupné z: `http://www.lidar-uk.com/index.php`.

[5] *VLP-16* [online] [cit. 2018-03-01]. Dostupné z: `http://velodynelidar.com/vlp-16.html`.

[6] WINKLER, Zbyněk. *Odometrie (Robotika.cz > Průvodce)* [online]. 2005 [cit. 2018-03-31]. Dostupné z: `https://robotika.cz/guide/odometry/cs`.

[7] BESL, P. J.; MCKAY, N. D. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1992, roč. 14, č. 2, s. 239–256. ISSN 0162-8828. Dostupné z DOI: `10.1109/34.121791`.

[8] BLANCO, Jose Luis. *Iterative Closest Point (ICP) and other matching algorithms – MRPT* [online]. 2013 [cit. 2018-04-15]. Dostupné z: `https://www.mrpt.org/Iterative_Closest_Point_%28ICP%29_and_other_matching_algorithms`.

[9] *Accelerometer, Gyro and IMU Buying Guide - SparkFun Electronics* [online] [cit. 2018-04-18]. Dostupné z: `https://www.sparkfun.com/pages/accel_gyro_guide`.

[10] STACHNISS, Cyrill. *Robot Mapping - WS 2013/14 - Arbeitsgruppe: Autonome Intelligente Systeme* [online] [cit. 2018-04-08]. Dostupné z: `http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/`.

[11] THRUN, Sebastian; BURGARD, Wolfram; FOX, Dieter. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series)*. The MIT Press, 2005. ISBN 0262201623. Dostupné také z: `https://www.amazon.com/Probabilistic-Robotics-Intelligent-Autonomous-Agents/dp/0262201623?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0262201623`.

[12] SOLA, Joan. *Towards visual localization, mapping and moving objects tracking by a mobile robot: a geometric and probabilistic approach*. 2007. Disertační práce. Institut National Polytechnique de Toulouse-INPT.

[13] MOUTARLIER, Philippe; CHATILA, Raja. An experimental system for incremental environment modelling by an autonomous mobile robot. In: *Experimental Robotics I*. 1990, s. 327–346.

[14] ANDRADE-CETTO, Juan; VIDAL-CALLEJA, Teresa; SANFELIU, Alberto. Unscented transformation of vehicle states in SLAM. In: *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. 2005, s. 323–328.

[15] NETTLETON, Eric W; DURRANT-WHYTE, Hugh F; GIBBENS, Peter W; GÖKTOGAN, Ali H. Multiple-platform localization and map building. In: *Sensor Fusion and Decentralized Control in Robotic Systems III*. 2000, sv. 4196, s. 337–348.

[16] THRUN, Sebastian; LIU, Yufeng; KOLLER, Daphne; NG, Andrew Y; GHAHRAMANI, Zoubin; DURRANT-WHYTE, Hugh. Simultaneous localization and mapping with sparse extended information filters. *The International Journal of Robotics Research*. 2004, roč. 23, č. 7-8, s. 693–716.

[17] STACHNISS, Cyrill. *EKF SLAM* [online] [cit. 2018-04-15]. Dostupné z: `http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam05-ekf-slam.pdf`.

[18] STACHNISS, Cyrill. *Short Introduction to Particle Filters and Monte Carlo Localization* [online] [cit. 2018-04-18]. Dostupné z: `http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam11-particle-filter.pdf`.

[19] MONTEMERLO, Michael; THRUN, Sebastian; KOLLER, Daphne; WEGBREIT, Ben. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In: *Eighteenth National Conference on Artificial Intelligence*. Edmonton, Alberta, Canada: American Association for Artificial Intelligence, 2002, s. 593–598. ISBN 0-262-51129-0. Dostupné také z: `http://dl.acm.org/citation.cfm?id=777092.777184`.

[20] MONTEMERLO, Michael; THRUN, Sebastian. FastSLAM 2.0. *FastSLAM: A scalable method for the simultaneous localization and mapping problem in robotics*. 2007, s. 63–90.

[21] GRISETTI, Giorgio; STACHNISS, Cyrill; BURGARD, Wolfram. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*. 2007, roč. 23, č. 1, s. 34–46.

[22] GRISETTI, Giorgio; KUMMERLE, Rainer; STACHNISS, Cyrill; BURGARD, Wolfram. A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine*. 2010, roč. 2, č. 4, s. 31–43.

[23] STACHNISS, Cyrill. *Least Squares* [online]. 2013 [cit. 2018-04-14]. Dostupné z: `http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam14-least-squares.pdf`.

[24] *ROS/Introduction - ROS Wiki* [online]. 2014 [cit. 2018-03-01]. Dostupné z: `http://wiki.ros.org/ROS/Introduction`.

[25] *ROS/Concepts - ROS Wiki* [online]. 2014 [cit. 2018-04-13]. Dostupné z: http://wiki.ros.org/ROS/Concepts.

[26] *Nodes - ROS Wiki* [online]. 2012 [cit. 2018-03-01]. Dostupné z: http://wiki.ros.org/Nodes.

[27] *Cartographer — Cartographer 1.0.0 documentation* [online] [cit. 2018-03-16]. Dostupné z: http://google-cartographer.readthedocs.io/en/latest/.

[28] HORNUNG, Armin; WURM, Kai M.; BENNEWITZ, Maren; STACHNISS, Cyrill; BURGARD, Wolfram. OctoMap: an efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*. 2013, roč. 34, č. 3, s. 189–206. ISSN 1573-7527. Dostupné z DOI: 10.1007/s10514-012-9321-0.

[29] RUSU, Radu Bogdan; COUSINS, Steve. 3D is here: Point Cloud Library (PCL). In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, 2011.

[30] *Cartographer ROS Integration — Cartographer ROS 1.0.0 documentation* [online] [cit. 2018-03-16]. Dostupné z: http://google-cartographer-ros.readthedocs.io/en/latest/.

[31] *Clock - ROS Wiki* [online]. 2015 [cit. 2018-03-16]. Dostupné z: http://wiki.ros.org/Clock.

[32] *rpng/android_sensors_driver: ROS Driver for Android Sensors (opencv3 and camera1 API)* [online] [cit. 2018-03-16]. Dostupné z: https://github.com/rpng/android_sensors_driver.

[33] *catkin/Tutorials/workspace_overlaying - ROS Wiki* [online]. 2018 [cit. 2018-04-17]. Dostupné z: http://wiki.ros.org/catkin/Tutorials/workspace_overlaying.

A Obsah přiloženého DVD

DVD obsahuje instalační skript `install.bash`, který stáhne a nainstaluje *ROS*, další potřebné balíčky do něj. Následně vytvoří workscape pro *Cartographer*, nastaví verzi *Cartographeru* na 0.3.0 a nakopíruje upravenou verzi *cartographer_ros*. Nakonec se vytvoří druhý workscape s balíčkem `obstacle_detection`. Poté co jsou tyto workscape připraveny a jejich balíčky nainstalovány, tak stačí si nastavit cesty do termínálu pomocí skriptu `vsb_ws/install/setup.bash`. Skript byl ověřen na čisté instalaci Ubuntu 16.04.

B První konfigurace pro spuštění Cartographeru

```
<launch>
    <param name="/use_sim_time" value="true" />

    <param name="robot_description" textfile="$(find cartographer_ros)/urdf/
       cart_test.urdf" />
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="
       robot_state_publisher" />
    <node name="cartographer_node" pkg="cartographer_ros" type="
       cartographer_node" args="
         -configuration_directory $(find cartographer_ros)/configuration_files
         -configuration_basename cart_test.lua" output="screen">
       <remap from="points2" to="velodyne_points" />
       <remap from="imu" to="android/tango1/imu" />
    </node>
    <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find
       cartographer_ros)/configuration_files/demo_3d.rviz" />
    <node name="playbag" pkg="rosbag" type="play" args="--clock $(arg
       bag_filename)" />
</launch>
```

Výpis 5: Konfigurace `.launch` souboru pro správné spuštění *Cartographeru* nad `bag` záznamem

```
<robot name="cart_test">
    <material name="orange">
        <color rgba="1.0 0.5 0.2 1" />
    </material>
    <material name="gray">
        <color rgba="0.2 0.2 0.2 1" />
    </material>
    <link name="imu_link">
        <visual>
            <origin xyz="0.0 0.0 0.0" />
            <geometry>
                <box size="0.06 0.04 0.02" />
            </geometry>
            <material name="orange" />
        </visual>
    </link>
    <link name="velodyne">
        <visual>
            <origin xyz="0.0 0.0 0.0" />
            <geometry>
                <cylinder length="0.07" radius="0.05" />
            </geometry>
            <material name="gray" />
        </visual>
    </link>
    <link name="base_link" />
    <joint name="imu_link_joint" type="fixed">
        <parent link="base_link" />
        <child link="imu_link" />
        <origin xyz="0 0 0" rpy="0 0 0" />
    </joint>
    <joint name="velodyne_joint" type="fixed">
        <parent link="base_link" />
        <child link="velodyne" />
        <origin xyz="0 0 0" rpy="0 0 0" />
    </joint>
</robot>
```

Výpis 6: Popis robota ve formátu urdf pro spuštění *Cartographeru*

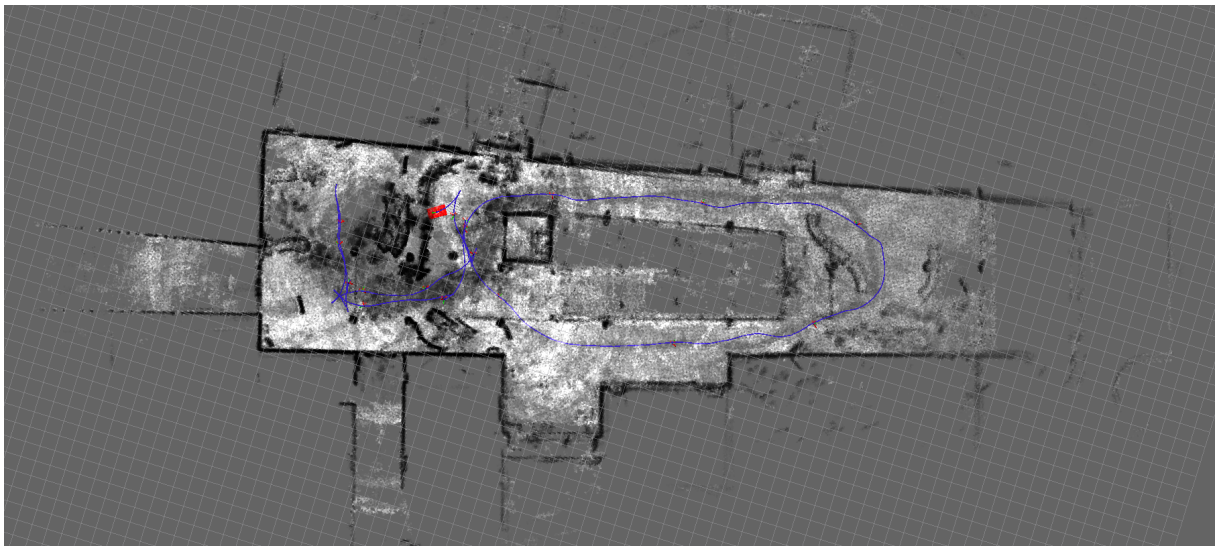
```
include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
  map_frame = "map",
  tracking_frame = "base_link",
  published_frame = "base_link",
  odom_frame = "odom",
  provide_odom_frame = true,
  use_odometry = false,
  num_laser_scans = 0,
  num_multi_echo_laser_scans = 0,
  num_subdivisions_per_laser_scan = 1,
  num_point_clouds = 1,
  lookup_transform_timeout_sec = 0.2,
  submap_publish_period_sec = 0.3,
  pose_publish_period_sec = 5e-3,
  trajectory_publish_period_sec = 30e-3,
  rangefinder_sampling_ratio = 1.,
  odometry_sampling_ratio = 1.,
  imu_sampling_ratio = 1.,
}

TRAJECTORY_BUILDER_3D.num_accumulated_range_data = 80

MAP_BUILDER.use_trajectory_builder_3d = true
MAP_BUILDER.num_background_threads = 3
POSE_GRAPH.optimization_problem.huber_scale = 5e2
POSE_GRAPH.optimize_every_n_nodes = 30
POSE_GRAPH.constraint_builder.sampling_ratio = 0.03
POSE_GRAPH.optimization_problem.ceres_solver_options.max_num_iterations = 10
POSE_GRAPH.constraint_builder.min_score = 0.62
POSE_GRAPH.constraint_builder.global_localization_min_score = 0.66

return options
```

Výpis 7: Konfigurace *Cartographeru*

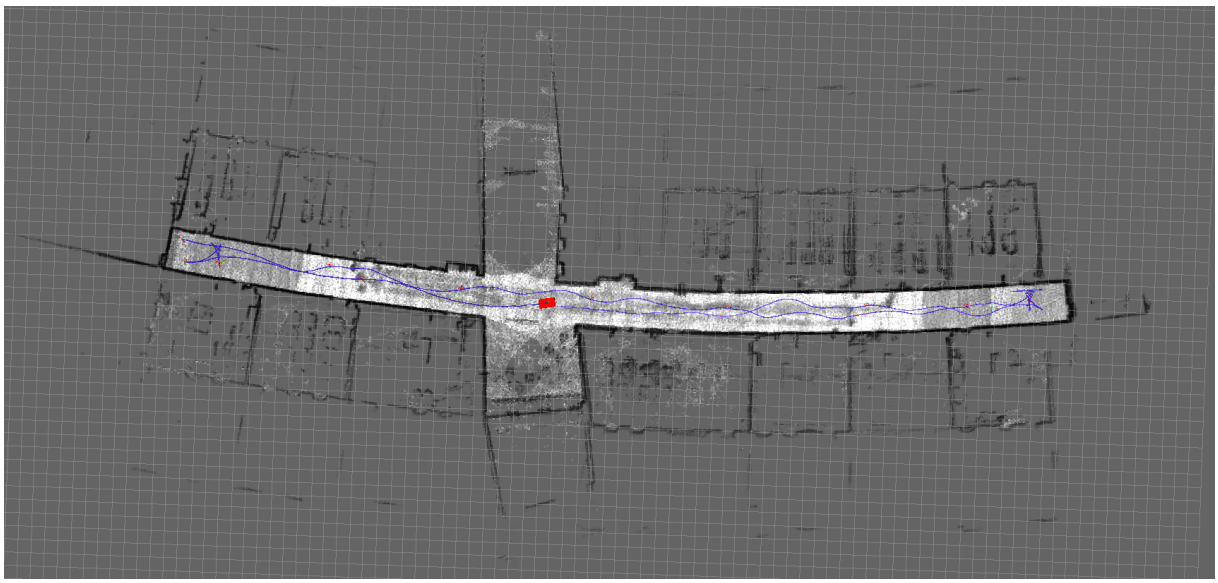
C Testovací prostředí

Obrázek 17: Fotka našeho auta, ze kterého byly pořizovány testovací data. Autor fotky: Mgr. Ing. Michal Krumnikl, Ph.D.

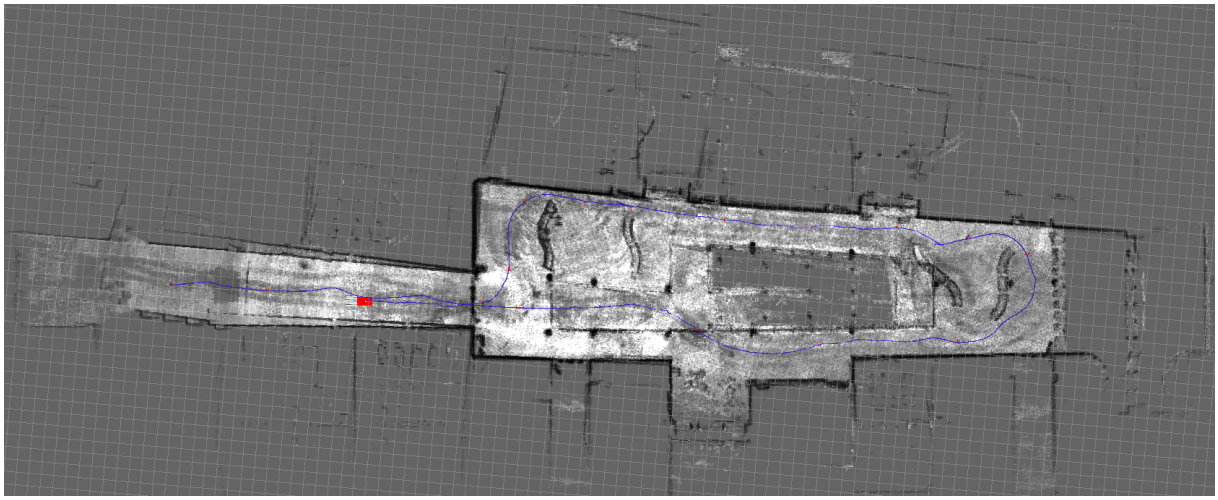
D Výsledky testovacích případů



Obrázek 18: Výsledek mapy při pohledu shora nad záznamem `dod.bag`



Obrázek 19: Výsledek mapy při pohledu shora nad záznamem `druhe_patro_1.bag`



Obrázek 20: Výsledek mapy při pohledu shora nad záznamem `druhe_patro_2.bag`